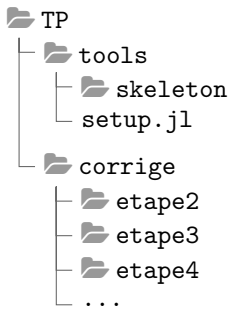
Julia : tour d'horizon

François Févotte, Laurent Plagne

Préambule

Ce TP a pour objectif de se familiariser avec le langage Julia, non seulement en apprenant les rudiments de sa syntaxe mais surtout en prenant de bonnes habitudes de développement. Nous nous concentrerons dans un premier temps sur la mise en place d'un environnement de travail adapté (et qui le restera dans la durée, y compris pour de plus gros projets). Dans un second temps, nous prendrons l'exemple d'un produit matrice-vecteur comme fil conducteur permettant de découvrir quelques aspects fondamentaux de Julia.

Cet énoncé de TP s'accompagne d'une archive de code source, organisée comme suit :



Le répertoire `corrige` contient un sous-répertoire pour chaque étape clé du TP. Pour chacune de ces étapes, correspondant à la fin de chaque partie de ce TP (et numérotées de la même manière), le sous-répertoire contient un ensemble de sources correspondant à la réalisation des questions du TP. En cas de doute ou de blocage durant ce TP, il est toujours possible de se référer au corrigé de l'étape en cours afin d'avoir un aperçu de la solution.

Le répertoire `tools` contient certains outils qu'il sera utile de mettre en œuvre durant le TP. Nous reviendrons sur chacun d'entre eux au moment opportun.

Toutes les questions numérotées devraient être réalisables à partir des informations données dans ce document (sinon : protestez !). Les questions "pour aller plus loin" sont, comme leur nom l'indique, plus difficiles et peuvent faire appel à des notions qui ne seront présentées que plus tard. A minima, il faut s'attendre à devoir lire la documentation Julia pour réaliser les développements proposés. Et ne pas hésiter à nous demander !

1 Installation préalable

Afin de préparer au mieux la session, nous recommandons aux participants d'installer quelques outils au préalable. Nous profitons de cette liste d'outils pour commencer à expliquer brièvement à quoi sert chacun d'entre eux, mais la suite de cette session reviendra sur les différentes notions avec plus de détails.

1.1 Julia

Julia s'installe simplement en téléchargeant une archive précompilée, ciblant une plate-forme (Linux, MacOS, Windows) et une architecture (32 ou 64 bits) particulières :

`https://julialang.org/downloads/`

Nous recommandons d'installer la dernière version disponible (v1.1.0), mais n'importe quelle version supérieure à v1.0.3 devrait être compatible.

1.2 Éditeur / IDE

Julia intègre uniquement un REPL, mais pas d'éditeur de texte ni d'environnement de développement intégré. Même si, a priori, n'importe quel éditeur de texte permet de travailler sur des sources Julia, et n'importe quel terminal permet d'interagir avec le REPL, nous suggérons ci-dessous quelques choix qui ont le mérite d'offrir un support spécifique de Julia :

Juno est un module basé sur l'éditeur de texte Atom. Il intègre de nombreuses fonctionnalités, y compris un terminal intégré permettant d'interagir avec le REPL Julia tout en éditant son code source. Il s'agit probablement de la meilleure solution "grand public", puisque la prise en main en est quasiment immédiate. Les instructions d'installation sont disponibles ici :

`http://junolab.org/`

Visual Studio Code propose une extension ajoutant du support pour le langage Julia. Il s'agit, comme Juno, d'une solution facile à prendre en main, mais le retour utilisateur semble en être un peu plus mitigé. Le développement du support Julia pour VSCode étant très actif, la situation devrait évoluer favorablement. Les instructions d'installation sont disponibles ici :

`https://marketplace.visualstudio.com/items?itemName=julialang.language-julia`

Emacs peut aussi être configuré pour fournir un grand nombre de fonctionnalités liées à Julia. Il s'agit toutefois d'une solution dont la mise en œuvre nécessite plus d'apprentissage sans expérience préalable avec Emacs. Les utilisateurs d'Emacs pourront en revanche adapter leur environnement en installant les paquets suivants (disponibles sur MELPA):

— `julia-mode` pour la coloration syntaxique
— `isend-mode` ou `julia-repl` pour l'interaction avec le REPL

Quelques détails supplémentaires sont donnés sur le forum *discourse* de Julia :

`https://discourse.julialang.org/t/emacs-based-workflow/19400`

1.3 Outils d'aide au développement en Julia

Quel que soit l'outillage "externe" choisi (éditeur de texte...), de nombreux outils d'aide au développement sont fournis sous forme de paquets Julia. Le script `tools/setup.jl` fourni avec ce TP permet d'en installer une sélection intéressante. La bonne exécution de ce script permet aussi de vérifier l'installation correcte de Julia.

Command line

```
$ path/to/setup.jl
[ Info: Logging debug information
    path = "/path/to/setup.log"
[ Info: Updating package versions
[ Info: Package `BenchmarkTools' already installed
[ Info: Package `Revise' already installed
[ Info: Package `Jive' already installed
[ Info: Package `Test' already installed
[ Info: Installing Retest
[ Info: Precompiling Retest
[ Info: Package `Profile' already installed
[ Info: Installing ProfileView
[...]
```

2 Premiers pas

2.1 Où trouver de l'aide

Documentation officielle : dans une première partie, le manuel de Julia aborde les grands aspects du langage. Une deuxième partie sert de documentation de référence pour la bibliothèque standard.

`https://docs.julialang.org/`

Un chapitre intéressant à la fin du manuel détaille quelques différences entre Julia et d'autres langages populaires, à l'attention des développeurs provenant d'une autre communauté :

`https://docs.julialang.org/en/v1/manual/noteworthy-differences/`

Documentation des paquets : il s'agit d'un répertoire listant l'ensemble des paquets Julia officiellement enregistrés, ainsi que la documentation associée :

`https://pkg.julialang.org/`

Forum *discourse* : il s'agit d'un accès privilégié à toute la communauté Julia. N'importe qui (du débutant au confirmé) peut y poser ses questions et y être toujours bien reçu. La qualité des réponses est impressionnante (mais parfois un peu déroutante quand une question simple suscite une vague de réponses proposant des approches différentes) :

`https://discourse.julialang.org/`

2.2 REPL

La première manière d'interagir avec Julia se situe dans le REPL (*Read Eval Print Loop*), qui peut se lancer dans n'importe quel terminal/console à l'aide de la commande `julia` (dans l'environnement Juno, un terminal est automatiquement présenté pour lancer Julia). En plus du mode par défaut, dans lequel on entre des commandes et en observe le résultat, le REPL Julia propose d'autres modes, comme l'aide. On active le mode d'aide en tapant `?` au début d'une ligne vide. Il est alors possible d'entrer le nom d'une commande pour en avoir la description et un exemple d'utilisation.

Command line

```
shell$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.1.0 (2019-01-21)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> 1+2
3

# Taper '?' pour entrer dans le mode 'help'
# puis le nom (potentiellement approximatif) de la commande cherchée, ici 'for'
help?> for
search: for foreach foldr floor mapfoldr factorial EOFError OverflowError

  for

  for loops repeatedly evaluate the body of the loop by iterating over a sequence of
  values.

  [...]

julia> for i = 1:3
         println(i)
       end
1
2
3
```

Question 1

Familiarisez vous avec le REPL en entrant quelques commandes. Utilisez l'aide en ligne pour explorer le langage (`for`, `if`, `println`, `struct`...).

2.3 Fichiers source

Si le REPL est très pratique pour tester de petites commandes, la réalisation de grands programmes nécessite de passer par des fichiers de code source (extension : `.jl`). Il est évidemment possible d'exécuter directement un programme julia depuis la ligne de commande :

`hello.jl`

```
n = 1 + 2
for i = 1:n
    println("Hello World!")
end
```

Command line

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

Question 2

Recopiez vos tests de la question précédente dans un fichier source Julia, et lancez-le depuis la ligne de commande.

Pour aller plus loin...

Y a-t-il des différences entre l'exécution dans le REPL et l'exécution dans un fichier ?

2.4 Interactions sources ↔ REPL

Cependant, ré-exécuter un programme Julia à chaque fois que ses sources changent s'avère fastidieux lors de la phase de développement. Même si ce n'est pas strictement nécessaire dans un premier temps, nous proposons de prendre de bonnes habitudes en mettant en place une méthodologie de travail plus interactive, qui restera adaptée aux grands projets.

2.4.1 Création d'un "projet"

Nous recommandons ainsi de toujours travailler dans un "projet", c'est-à-dire un ensemble de fichiers sources suivant l'organisation préconisée par le gestionnaire de paquets Julia. L'outil `skeleton.jl` permet de simplifier la création de telles arborescences de sources. Cet outil est un simple script Julia, dont une version à jour vous est livrée dans les sources de ce TP. Sinon, il est toujours possible de l'installer en le "clonant" depuis github:

`https://github.com/ffevotte/skeleton.jl`

NB : Si votre environnement git est correctement configuré, `skeleton.jl` l'utilisera pour en déduire certaines informations :

- `user.name` : nom du développeur principal
- `user.email` : adresse e-mail
- `github.user` : nom d'utilisateur github

Ces données ne sont nécessaires que pour une utilisation plus avancée des paquets (intégration continue dans Github, diffusion officielle dans les dépôts Julia...) mais la configuration de git (à l'aide de la commande `git config`) n'est pas non plus un investissement insurmontable. Par exemple :

```
$ git config -g user.name "Prénom Nom"
$ git config -g user.email "prenom.nom@domaine.ext"
$ git config -g github.user "ghuser"
```

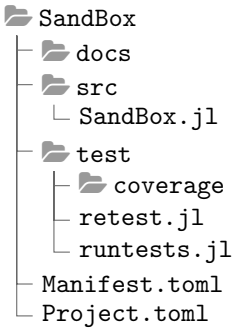
Question 3

(a) Utilisez `skeleton.jl` pour créer un projet vierge. Vous devriez obtenir une arborescence de fichiers similaire à celle montrée ci-dessous.

(b) Explorez un peu les différents fichiers créés et leur contenu.

Command line

```
$ path/to/tools/skeleton/skeleton.jl SandBox
[ Info: skeleton.jl
    Julia_VERSION = v"1.1.0"
    githash = "ab609a7a54e9e37cf14e42fbc7e16a4dd696d983"
[ Info: copy and substitute
    srcdir = "/path/to/skeleton.jl/template"
    destdir = "SandBox"
[...]
```



Sans rentrer dans les détails, voici les quelques points à retenir sur cette structure de fichiers :

— `Project.toml`: la simple présence de ce fichier signale à Julia qu'il s'agit d'un "projet". Par ailleurs, ce fichier contient aussi des méta-informations importantes sur le projet lui-même ;

— `src/SandBox.jl`: c'est dans le répertoire `src` que se trouvent les sources du projet. En particulier, un fichier "principal" portant le nom du projet (ici `SandBox.jl`) sert de point d'entrée pour le reste des sources ;

— `test/runtests.jl`: c'est dans le répertoire `test` que se trouvent les sources Julia permettant de tester le code du projet. En particulier, un fichier `runtests.jl` sert de point d'entrée pour le lancement de tous les tests ;

Nous devinons par ailleurs que l'utilisation de `skeleton.jl` a aussi permis de préparer les étapes suivantes dans la vie du projet : documentation (répertoire `docs`), intégration continue et couverture de tests (répertoire `test/coverage`). Nous ne nous intéresserons pas à ces sujets dans un premier temps. Pour l'instant, nous nous concentrons sur la mise en place de méthodes de développement interactives, permettant à la fois de capitaliser et structurer le code source dans des fichiers, et de bénéficier d'une interaction rapide avec le REPL.

2.4.2 Développement interactif

Le programme `test/retest.jl` permet de mettre en place un tel *workflow* interactif, basé sur les paquets `Revise`, `Jive` et `Retest`. En exécutant dans un terminal le programme `retest.jl`, on ouvre un REPL interactif :

Command line

```
bash$ cd /path/to/SandBox/test
bash$ ./retest.jl
watching folders ...
  - ../src
  - .
  - coverage
[ Info: Precompiling SandBox [26d708fe-1ec2-56ce-9e97-fa6abf607293]
julia>
```

Dans le cas de Juno où un REPL Julia fonctionne déjà dans la console, il suffit d'entrer les commandes suivantes. On peut aussi lancer Juno depuis le répertoire de test pour pouvoir utiliser des chemins relatifs plus simples.

```
julia> pwd()
"/path/to/current/directory"

julia> cd("/path/to/SandBox/test/")

julia> include("retest.jl")
```

On peut interagir avec ce REPL de manière habituelle. Mais ce REPL exécute aussi, à chaque fois qu'un fichier source est mis à jour, toutes les commandes se trouvant dans le fichier `test/runtests.jl`:

SandBox/test/runtests.jl

```
using SandBox
using Test
using Retest        # @itest

for i in 1:3
    println("Hello World!")
end

@itest begin
    1 + 1
end
```

Command line

```
[ Info: File changed
[   path = "runtests.jl"
Hello World!
Hello World!
Hello World!

Interactive test:
quote
    1 + 1
end

2
```

NB : Lorsqu'une commande est mise dans un bloc `@itest`, le résultat produit est affiché en dessous, de la même manière que si l'expression avait été entrée dans un REPL (sans point-virgule pour supprimer l'affichage).

Nous entrerons ultérieurement dans le détail du fonctionnement des paquets et modules, ce qui nous permettra alors de séparer le code source en bibliothèques de fonctions (les modules) et tests appelant ces fonctions. Mais dans un premier temps, nous ne structurerons pas autant notre code et nous contenterons de travailler dans le fichier `runtests.jl`.

Question 4

(a) Lancez le REPL interactif à l'aide du script `retest.jl`.

(b) Entrez quelques commandes dans le fichier `runtests.jl` pour constater la mise à jour des sorties du code à chaque changement (sauvegardé) des sources.

3 Variables et fonctions

Julia est un langage fondé sur le principe du dispatch multiple (*multiple dispatch*), dans lequel deux ingrédients jouent un rôle fondamental :

— les types, et
— les fonctions ou, comme on le verra par la suite, plus précisément les méthodes.

3.1 Variables, valeurs et types

Chaque valeur manipulée par Julia possède un *type*, décrivant sa nature. Dans une première approche, voici quelques uns des types que nous serons amenés à manipuler :

Command line

```
# Un nombre entier (stocké ici sur 64 bits en raison de l'architecture de la machine)
julia> 10
10

julia> typeof(10)
Int64

# Un nombre a à virgule flottante, stocké sur 64 bits
julia> 3.14
3.14

julia> typeof(3.14)
Float64

# Un tableau (unidimensionnel) de valeurs entières
julia> [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

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

La gestion des variables en Julia suit des conventions assez répandues [1]. Une variable est simplement un "nom" associé à une valeur. Réaffecter la variable change simplement l'association nom ↔ valeur, sans modifier la valeur elle-même. Seules les valeurs sont typées ; les variables elles-mêmes ne le sont pas [2] : il est tout à fait possible de réaffecter une variable en lui assignant une nouvelle valeur d'un autre type :

Command line

```
# Association du nom "x" à la valeur 10
julia> x = 10
10

# Association du nom "y" à la valeur 10
julia> y = x
10

julia> typeof(x)
Int64

# Réaffectation de la variable à une
# nouvelle valeur
julia> x += 1.1      # <=> x = x + 1.1
11.1

julia> typeof(x)
Float64

# "y" n'a jamais été réaffecté
#   => pas de changement de valeur
julia> y
10
```

Command line

```
# Association de "x" à un tableau
julia> x = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

# Association de "y" au même tableau
julia> y = x
3-element Array{Int64,1}:
 1
 2
 3

# Mutation du contenu du tableau
julia> x[1] = 4  # <=> setindex(x, 4, 1)
4

# "y" a changé aussi
julia> y
3-element Array{Int64,1}:
 4
 2
 3
```

3.2 Fonctions

Les fonctions constituent l'unité de base pour l'organisation du code source en Julia. Une fonction peut se définir de deux manières différentes :

```
# Version longue
function fun_name(arg1, arg2)
    tmp = arg1 + arg2
    tmp * arg1          # <- le mot-clé ``return'' est optionnel :
end                     #    la dernière valeur calculée est retournée
                        #    comme résultat de la fonction


# Version courte
fun_name(arg1, arg2) = arg1 * (arg1 + arg2)
```

et la syntaxe d'appel est très standard:

```
julia> fun_name(2, 3)   # Attention, espace interdit entre le nom de la
                        # fonction et la parenthèse ouvrante
10
```

1. qu'on retrouve par exemple dans des langages comme Python
2. contrairement à des langages comme C ou C++

Question 5

Implémentez (dans le fichier `SandBox/test/runtest.jl`) la fonction `my_gcd(a, b)` réalisant le calcul du PGCD de deux nombres `a` et `b`. On pourra utiliser par exemple l'algorithme d'Euclide (Alg. 1).

Exemple de test :

```
@test my_gcd(1386, 3213) == 63
```

Sujets utiles (à rechercher dans la documentation pour plus de détails) :
- `%`, `mod`, `divrem`, `div` : quotient et reste dans la division entière ;
- `for`, `while` : boucles ;
- `==`, `!=`, `<`, `>`, *etc.* : opérateurs de comparaison.

Algorithm 1: calcul de PGCD (Euclide)

Input: $(a, b) \in \mathbb{N}^2$

while $b \neq 0$ **do**
  Déterminer $(q, r) \in \mathbb{N}^2$ tels que $a = b\,q + r$;
  $a \leftarrow b$;
  $b \leftarrow r$;
end
return a ;

Pour aller plus loin...

(a) La valeur des arguments (`a` et `b`) est-elle modifiée à l'extérieur de la fonction ?

(b) Récrire la fonction `my_gcd` de plusieurs manières différentes. Par exemple : de manière récursive, avec terminaison prématurée (`return`) en cas de reste nul...

3.3 Portée des variables

La portée des variables suit des conventions relativement classiques puisque Julia applique des règles de portée lexicale (*lexical scoping*) similaires à celles qu'on retrouve dans quasiment tous les langages de programmation [3]. Dans le détail :
- Une variable nouvellement introduite dans une portée sera héritée dans toutes les sous-portées, mais ne sera en revanche pas visible dans les portées englobantes.
- Une nouvelle portée globale est introduite pour le REPL, ainsi que pour chaque module. Les portées globales sont entièrement indépendantes les unes des autres, et n'ont pas de portée englobante.
- Une nouvelle portée locale est introduite pour le corps des boucles (`for`, `while`) ou des fonctions (`function`) ainsi que les blocs de traitement d'exception (`try...catch...finally`) et la commande spécifique `let`.
- Les blocs `if` ou `begin` n'introduisent pas de nouvelle portée.

Attention : Si une variable est définie dans la portée globale, elle n'est pas héritée en écriture par ses sous-portées. Cette entorse à la règle répond à des exigences de performance, et n'est en pratique visible que pour des démonstrations dans le REPL.

La convention de passage d'arguments est la suivante : à l'intérieur de la fonction, l'argument est considéré comme une variable locale, initialisée avec la valeur (non copiée) donnée lors de l'appel. Une réaffectation dans la fonction change simplement l'association variable ↔ valeur (aucune modification visible dans le contexte de l'appel de fonction), tandis qu'une mutation de la valeur de la variable à l'intérieur de la fonction aura un impact potentiellement visible dans le contexte de l'appel. Cette convention, aussi employée par des langages comme Python, est parfois appelée *pass by sharing*.

Par convention, les fonctions modifiant au moins un de leurs arguments portent un nom se terminant par le caractère "`!`".

3. à l'exception peut-être de certains dialectes LISP utilisant une portée dynamique (*dynamic scoping*).

Question 6 : quelques fonctions d'algèbre linéaire

(a) Implémentez la fonction `my_axpy!(a, X, Y)`, qui remplace le vecteur `Y` par `a*X + Y`, où `a` est un scalaire et `X` et `Y` sont des vecteurs. La nouvelle valeur de `Y` est renvoyée.

Exemple de test :

```
@test begin
    X = [1, 2, 3]
    Y = [4, 5, 6]
    a = 100
    my_axpy!(100, X, Y)
    Y == [104, 205, 306]
end
```

(b) Implémentez la fonction `my_sum(X)` qui calcule la somme des éléments du vecteur `X`.

Exemple de test :

```
@test begin
    X = [1, 2, 3]
    my_sum(X) == 6
end
```

(c) Implémentez la fonction `prodMV(M, V)` qui calcule le produit de la matrice `M` par le vecteur `V`.

Exemple de test :

```
@test begin
    M = [1 2; 3 4; 5 6]
    V = [7, 8]
    prodMV(M, V) == [23, 53, 83]
end
```

Informations utiles :

- un vecteur peut être défini explicitement à l'aide de la syntaxe `X = [1, 2, 3]`,
- une matrice peut être définie explicitement à l'aide de la syntaxe `M = [1 2; 3 4]`,
- vecteurs et matrices sont des cas particuliers (1D ou 2D) de vecteurs multidimensionnels, qui peuvent être créés en Julia grâce de nombreuses fonctions : `zeros`, `ones`, `rand`...
- par défaut, l'indexation des éléments commence à 1 et l'accès se fait grâce à la syntaxe `X[i]`, `M[i,j]`, ou plus généralement `V[i,j,k,l...]` pour un vecteur multidimensionnel,
- la taille des vecteurs et matrices est donnée par les fonctions : `size` ou `length`.

Pour aller plus loin...

Expérimentez les subtilités de la portée des variables en recopiant *le corps* de la fonction `my_sum` directement dans la portée globale.

4 Compilateur *Just Ahead of Time* et *benchmarking*

Afin d'observer le comportement du compilateur Julia, une première approche consiste à mesurer les temps d'exécution de nos fonctions. Une première manière de faire utilise la macro `@time` de la bibliothèque standard:

Command line

```
julia> x = rand(Float64, 1_000_000);

julia> @time my_sum(x)
  0.040405 seconds (14.77 k allocations: 779.895 KiB)
499969.99243794015

julia> @time my_sum(x)
  0.001707 seconds (5 allocations: 176 bytes)
499969.99243794015

julia> @time my_sum(x)
  0.002787 seconds (5 allocations: 176 bytes)
499969.99243794015
```

Question 7

Réaliser plusieurs mesures de temps pour les fonctions que vous avez écrites, dans des conditions identiques (même type de données, même taille de problème).

Pour aller plus loin...

D'après vos essais, à quelle(s) condition(s) obtient-on une exécution anormalement lente ?

On remarque ici que la première exécution de la fonction prend plus de ressources (temps, allocations mémoire) que les suivantes. Cette première exécution est perturbée par le compilateur Julia, qui produit un code optimisé pour le type de données fourni à la fonction. Lors des exécutions suivantes, le code déjà compilé est réutilisé ; on ne mesure plus que l'exécution de la fonction proprement dite. Ce type de compilation au dernier moment est parfois qualifié de *Just Ahead Of Time* (JAoT).

Lors d'une utilisation réelle du programme, on peut considérer (ou espérer) que le temps de compilation sera négligeable devant le temps d'exécution. Le paquet `BenchmarkTools` permet de mieux évaluer le temps d'exécution seul, en éliminant les artefacts liés à la compilation :

Command line

```
julia> using BenchmarkTools

julia> @btime my_sum($x)
  1.455 ms (0 allocations: 0 bytes)
499969.99243794015

julia> @btime my_sum($x)
  1.455 ms (0 allocations: 0 bytes)
499969.99243794015
```

Notez que les arguments de fonctions doivent être préfixés par le caractère "`$`" lors des appels à la macro `@btime`.

Question 8

Réalisez à nouveau les mesures de temps de la question 7, en utilisant plutôt la macro `@btime`.

Pour aller plus loin...

En termes de performances, comment se compare votre implémentation de `my_sum` par rapport à la fonction standard `sum` ? Faites les comparaisons à la fois pour des vecteurs de nombres flottants (`rand(Float64, n)`) et de nombres entiers (`rand(1:100, n)`), pour `n` de l'ordre de 10^6.

En attendant le dernier moment pour compiler la fonction, le compilateur JAoT s'assure de disposer de la quantité maximale d'informations pour optimiser le code généré : type précis des variables, valeurs des arguments constants le cas échéant, *etc.*

Question 9

Une implémentation possible de la fonction `my_sum` est donnée par :

```
function my_sum(X)
    acc = 0
    for x in X
        acc += x
    end
    acc
end
```

(a) Quelle est la classe de complexité algorithmique de cette fonction ?

(b) Mesurez la classe de complexité de l'implémentation ? Est-ce cohérent ?

```
x1 = rand(1:10, 1_000_000)
x2 = rand(1:10, 2_000_000)
@btime my_sum($x1)
@btime my_sum($x2)
```

(c) Que se passe-t-il lorsqu'on appelle `my_sum(1:n)` ?

```
@btime my_sum(1:1_000_000)
```

Pour aller plus loin...

Il est possible d'étudier le code généré par le compilateur à l'aide des macros `@code_llvm` ou `@code_native`. Regardez quel code est produit par Julia pour la sommation des n premiers nombres entiers :

```
@code_native my_sum(1:10)
```

Et lorsqu'on remplace les vecteurs (`Array`) par des n-uplets (`Tuple`) ?

```
arr = [1, 2, 3]
@code_native my_sum(arr)

tup = (1, 2, 3)
@code_native my_sum(tup)
```

5 Types et dispatch multiple

Nous venons de mettre en évidence une première interaction entre les fonctions et les types : un même algorithme, tel qu'implémenté dans une fonction Julia, sera recompilé pour produire un code optimal pour chaque type de données sur lequel il est amené à opérer. Ceci est très important pour obtenir de la performance tout en évitant de dupliquer du code, mais il arrive fréquemment que ce mécanisme ne suffise pas : il devient nécessaire, pour implémenter une même opération pour des types de données différents, d'utiliser des algorithmes différents. C'est ici que les méthodes entrent en jeu.

5.1 Méthodes

Une *fonction* Julia n'est en réalité qu'un nom, associé à une sémantique particulière. Par exemple, la fonction `+` a la sémantique d'une loi de composition interne additive dans un anneau. Une *méthode* est la spécification de l'implémentation d'une fonction pour un type de données particulier. Dans l'exemple de la fonction `+`, on peut définir une méthode agissant sur les entiers de 64 bits (il s'agit alors de l'addition modulaire dans $\mathbb{Z}/2^{64}\mathbb{Z}$). On peut aussi définir une méthode travaillant sur des flottants en double précision et implémentant l'addition au sens de la norme IEEE-754. Il en va de même pour les nombres complexes, les matrices, *etc.*

Il est possible (mais pas nécessaire) de créer une fonction générique "nue" en Julia en omettant la liste des arguments et le corps de la fonction :

Command line

```
julia> function f end
f (generic function with 0 methods)
```

Les méthodes sont définies à l'aide de la syntaxe vue dans les chapitres précédents. Si une méthode est créée sans que la fonction correspondante ait été préalablement définie, celle-ci est créée automatiquement. Les méthodes se différencient entre elles par le nombre de leurs arguments ainsi que le type de ceux-ci, spécifié à l'aide de l'opérateur `::`. L'absence de spécification du type d'un argument est équivalente à une spécification `::Any`, c'est-à-dire que la méthode tolère des arguments de n'importe quel type.

Command line

```
julia> f(x) = "Méthode unaire générique : $x"
f (generic function with 1 method)

julia> f(x::Integer) = "Méthode unaire spécifique (Integer) : $x"
f (generic function with 2 methods)

julia> f(x, y) = "Méthode binaire générique : $x, $y"
f (generic function with 3 methods)

julia> f(x::Real, y) = "Méthode binaire spécifique (Real, Any) : $x, $y"
f (generic function with 4 methods)

julia> methods(f)
# 4 methods for generic function "f":
[1] f(x::Integer) in Main at REPL[2]:1
[2] f(x::Real, y) in Main at REPL[4]:1
[3] f(x) in Main at REPL[1]:1
[4] f(x, y) in Main at REPL[3]:1
```

Attention : Chaque nouvelle méthode vient s'ajouter à la liste des précédentes. Une nouvelle méthode ne remplace une implémentation précédemment définie que si le nombre et le type des arguments sont rigoureusement identiques.

Ceci peut provoquer des surprises avec le processus de développement interactif que nous avons adopté : supprimer une méthode des sources ne la fait pas disparaître du REPL. De même, modifier la liste des arguments d'une méthode ne fait pas disparaître l'ancienne version.

Lorsque nous adopterons une démarche reposant complètement sur les modules, ces effets de bord pourront être largement réduits. En attendant, il s'agit d'un point à garder à l'esprit.

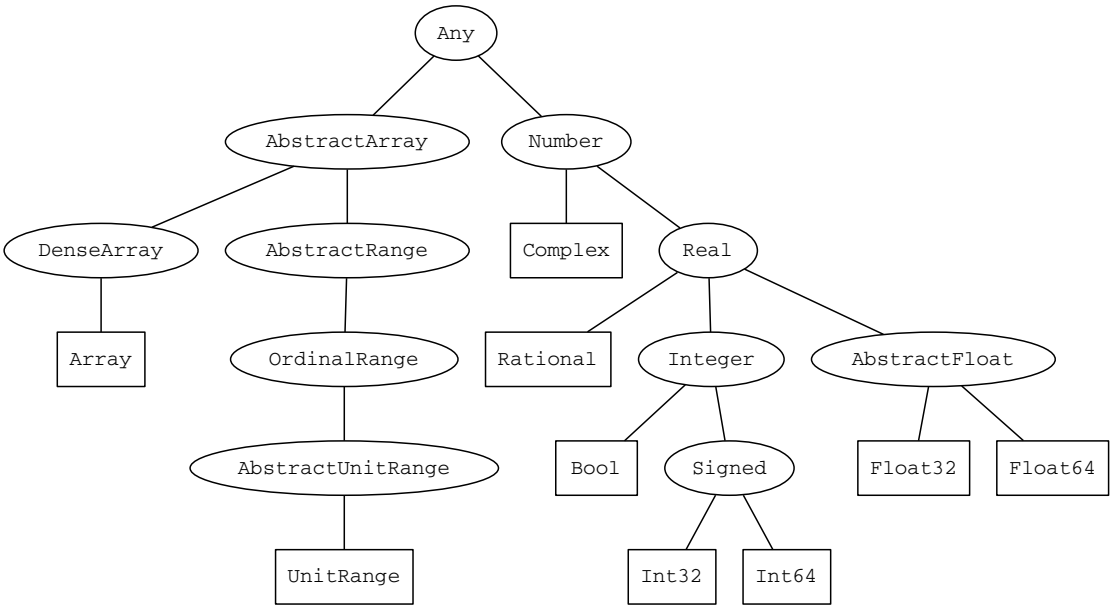


FIGURE 1 – Une partie de la hiérarchie des types

Lors d'un appel de fonction, la méthode particulière à utiliser est choisie en fonction du nombre et du type de valeurs fournies comme arguments. Ce choix s'appuie sur une hiérarchie de types : l'ensemble des types définis en Julia est organisé dans une structure d'arbre, dont la racine est le type `Any`. Chaque type est un nœud de l'arbre et possède un parent unique (son *supertype*), et un ensemble (potentiellement vide) d'enfants (les *sous-types*). La figure 1 représente une partie de la hiérarchie de types de Julia, centrée sur les nombres. Les feuilles de l'arbre (*i.e.* les types sans sous-type) y sont représentés par des nœuds rectangulaires. Il s'agit des types *concrets* de Julia, pour lesquels il est possible d'instancier des valeurs. Les nœuds internes de l'arbre sont des types *abstraits*, qui ne servent qu'à organiser leurs sous-types.

Une première manière de penser les relations supertype → sous-type dans la hiérarchie des types est de les considérer comme modélisant des relations ensemble → sous-partie pour les valeurs associées. Ceci est particulièrement vrai pour les types abstraits ; moins pour les types concrets. Par exemple, toutes les valeurs possibles pour un `Int32` sont aussi représentables en tant que valeurs du type `Int64`. Pour autant, `Int32` n'est pas un sous-type de `Int64`. Ainsi, il vaut mieux penser la hiérarchie de types en termes de comportement : lorsqu'un comportement est défini pour un supertype, il devrait avoir un sens pour toutes les valeurs des sous-types.

Une méthode définie pour accepter des arguments d'un type abstrait (par exemple `Real`), est valide pour des arguments de n'importe quel sous-type (par exemple `Float64` ou `Bool`). Lors de l'appel d'une fonction, le type (concret) des arguments est connu ; la méthode valide la plus spécifique est choisie :

Command line

```
julia> f(pi)
"Méthode unaire générique : pi = 3.1415926535897..."

julia> f(1)
"Méthode unaire spécifique (Integer) : 1"

julia> f(im, 2)
"Méthode binaire générique : im, 2"

julia> f(pi, [1, 2, 3])
"Méthode binaire spécifique (Real, Any) : pi = 3.1415926535897..., [1, 2, 3]"

julia> @which f(42)
f(x::Integer) in Main at REPL[2]:1
```

Question 10

(a) Que se passe-t-il si la fonction `prodMV` est appelée avec des arguments de types inattendus ? Par exemple :

```
prodMV(2, 3)
prodMV(true, false)
prodMV("hello", "world!")
```

(b) Révisez votre implémentation de `prodMV` afin qu'une erreur soit renvoyée si les arguments ne sont pas des tableaux. Comportement attendu :

```
prodMV(2, 3)                # -> erreur
prodMV(true, false)         # -> erreur
prodMV("hello", "world")    # -> erreur
prodMV([1 2; 3 4], [5, 6]) # -> [17, 39]
```

Pour aller plus loin...

(a) Que se passe-t-il pour `prodMV([1, 2], [3, 4])` ?

(b) Le problème doit-il être réglé par des vérifications dans le code ? Pourrait-il être détecté directement par le système de types ?
Indication : on pourra regarder quels sont les types respectifs des valeurs `[1 2; 3 4]` et `[1,2]`.

(c) Proposez une solution permettant d'obtenir le comportement suivant, dans lequel les erreurs sont générées directement par le compilateur sur la base du système de types :

```
prodMV(2, 3)                # -> erreur
prodMV(true, false)         # -> erreur
prodMV("hello", "world")    # -> erreur
prodMV([1,2], [3,4])        # -> erreur
prodMV([1 2; 3 4], [5, 6]) # -> [17, 39]
```

5.2 Types composites

Nous venons de voir une première manière de s'appuyer sur le dispatch multiple : définir de nouvelles fonctions et implémenter des méthodes spécialisées pour un certain nombre de types (déjà existants). Il est aussi possible d'adopter une démarche complémentaire, en définissant de nouveaux types, et les méthodes permettant d'implémenter pour ces types des fonctions déjà existantes.

Les types abstraits se définissent très simplement de la manière suivante. Par exemple, pour définir un nouveau type abstrait `MyAbstractComplex` dont le super-type est `Number` :

```
abstract type MyAbstractComplex <: Number end
```

L'immense majorité des types concrets que nous serons amenés à définir sont des types *composites* (souvent appelés enregistrements, structures ou objets dans d'autres langages). Un type composite est une collection de champs nommés, et optionnellement (mais il est très fortement recommandé de le faire) typés :

```
struct MyComplex <: MyAbstractComplex
    real :: Float64
    imag :: Float64
end
```

Une fonction, portant le même nom que le type composite et prenant comme arguments les champs nommés, permet de créer des instances du type. On accède à la valeur des champs d'une instance avec la syntaxe `instance.field` :

Command line

```
julia> c = MyComplex(1, 3)
MyComplex(1.0, 3.0)

julia> c.real
1.0

julia> c.imag
3.0
```

Il est évidemment possible de définir de nouvelles fonctions agissant sur un type nouvellement créé. Mais il est aussi très intéressant d'étendre des fonctions existantes en implémentant des méthodes spécifiques pour le type nouvellement créé. Ceci est particulièrement vrai pour l'extension des fonctions standard de Julia (définies dans le module `Base`).

```
import Base.+
function +(a::MyComplex, b::MyComplex)
    MyComplex(a.real+b.real, a.imag+b.imag)
end
```

Attention : Il est très déconseillé d'étendre des fonctions déjà existantes (*i.e.* définies par d'autres que vous) en y ajoutant des méthodes travaillant sur des types déjà existants (*i.e.* définis par d'autres que vous). Ce type de comportement est désigné sous le nom de "piratage de type" (***type piracy***).

Afin d'éviter le piratage de type par inadvertance il est nécessaire d'importer explicitement les fonctions que l'on souhaite étendre avant de leur définir de nouvelles méthodes. C'est le sens de la commande "`import Base.+`" ci-dessus. Nous reviendrons plus tard sur les détails des modules.

Ces précautions étant prises, l'extension des fonctions standard permet d'obtenir d'excellentes propriétés de polymorphisme. Du point de vue de l'addition, notre type `MyComplex` se manipule maintenant exactement de la même manière que n'importe quel type standard du langage :

Command line

```
julia> a = MyComplex(1, 2);
julia> b = MyComplex(3, 4);
julia> a + b
MyComplex(4.0, 6.0)
```

Question 11

(a) Implémentez votre propre type de nombres rationnels.

(b) Définissez quelques opérations arithmétiques de base sur ce type.

Pour aller plus loin...

(a) Améliorez votre type au point d'être en mesure de faire fonctionner vos implémentations `my_axpy!`, `my_sum` et `prodMV` avec cette arithmétique rationnelle. On pourra aussi re-définir la méthode `show`, permettant d'afficher une valeur rationnelle de manière plus lisible.

(b) Même question avec les fonctions standard d'algèbre linéaire : `dot`, résolution de systèmes avec l'opérateur "\", etc.

Pointeurs :
— `promote_rule` pour définir les règles de promotion `Int`→`MyRational` ;
— `BigInt` pour éviter les *overflows* d'entiers dans les applications réelles.

5.3 Types paramétrés

Nous avons déjà été amenés à manipuler des types comme `Array{Int64, 2}`, qui représentent des tableaux bi-dimensionnels à éléments entiers :

Command line

```
julia> typeof([1 2; 3 4])
Array{Int64,2}
```

IL s'agit là de types *paramétrés* : la définition du type est connue du compilateur, mais fait apparaître des paramètres libres qui seront connus uniquement au moment de l'instanciation.

Julia fournit une syntaxe particulière permettant de définir des types paramétrés, ainsi que les méthodes qui agissent sur ces types :

```
struct MyComplex{T} <: MyAbstractComplex
    real :: T
    imag :: T
end

# Inutile de préciser que le type est paramétré si on n'a pas besoin de
# retrouver le paramètre
real(x::MyComplex) = x.real

# La syntaxe "where" permet de préciser que "T" est un paramètre
function Base.zero(_::MyComplex{T}) where {T}
    MyComplex{T}(zero(T), zero(T))
end
```

Un constructeur par défaut est créé pour le type, qui sélectionne les paramètres en fonction des arguments qui lui sont passés. Sinon, il est toujours possible de spécifier l'ensemble des paramètres à la construction, auquel cas le compilateur mettra en place une promotion des arguments dans les bons types. À part pour ces détails de construction, les instances de types paramétrés se manipulent exactement comme toutes les autres.

Command line

```
julia> x = MyComplex(1, 2)
MyComplex{Int64}(1, 2)

julia> real(x)
1

julia> base_type(x)
Int64
```

Command line

```
julia> y = MyComplex{Float64}(1, 2)
MyComplex{Float64}(1.0, 2.0)

julia> real(y)
1.0

julia> base_type(y)
Float64
```

Pour aller plus loin...

(a) Changez la définition de votre fonction `prodMV(M, V)` pour qu'elle accepte comme argument n'importe quel tableau bidimensionnel `M` et monodimensionnel `V`, quel que soit le type de leurs éléments.

(b) Transformez votre type `MyRational` en le paramétrant par le type de son numérateur / dénominateur.

6 Architecture logicielle et qualité de code

6.1 Modules

Les modules constituent en Julia le moyen d'organiser sémantiquement les types et fonctions (et variables globales) dans des "espaces de travail" ou "espaces de noms" (*namespaces*) séparés.

```
module Polygons
  struct Polygon end
  draw(p::Polygon) = "drawing polygon"
end

module Circles
  struct Circle end

  # cette fonction ``draw'' n'est pas la même que celle du module ``Polygons''
  draw(c::Circle) = "drawing circle"
end

p = Polygons.Polygon()
Polygons.draw(p)

c = Circles.Circle()
Circles.draw(c)
```

Pour alléger le code utilisant le module, il est possible que celui-ci "exporte" des noms, qui seront alors disponibles dans le code utilisateur sans préciser dans quel module on les trouve :

```
module Polygons
  export Polygon

  struct Polygon end
  draw(p::Polygon) = "drawing polygon"
end

# Déclaration d'utilisation du module ``Polygons''
using .Polygons

# Les noms exportés par ``Polygons'' sont maintenant disponibles sans
p = Polygon()

# Les noms non exportés peuvent toujours être utilisés à condition d'être qualifiés
Polygons.draw(p)
```

Les modules peuvent être imbriqués les uns dans les autres :

```
module Shapes

  module Polygons
    export Polygon, draw

    struct Polygon end
    draw(p::Polygon) = "drawing polygon"
  end

  module Circles
    export Circle, draw

    struct Circle end
    draw(c::Circle) = "drawing circle"
  end
end

# Utilisation du sous-module ``Polygons'' défini dans ``Shapes''
using .Shapes.Polygons
```

```
p = Polygon()
draw(p)
```

Si deux modules exportent le même nom, il y a conflit:

```
using .Shapes.Circles # WARNING: using Circles.draw in module Main conflicts
                      # with an existing identifier.
c = Circle()
draw(c)               # ERROR: MethodError: no method matching draw(::Circle)
Circles.draw(c)       # OK
```

Dans le cas des fonctions/méthodes, une manière de résoudre le conflit consiste à fusionner les définitions concurrentes comme méthodes de la même fonction :

```
module Shapes
  export draw

  # Une fonction sans aucune méthode
  function draw end

  module Polygons
    export Polygon, draw
    struct Polygon end

    # la fonction "draw" devient visible
    using ..Shapes

    # qualification explicite nécessaire pour ajouter une méthode
    Shapes.draw(p::Polygon) = "drawing polygon"
  end

  module Circles
    export Circle, draw
    struct Circle end

    # import explicite de "draw" depuis le module "Shapes"
    import ..Shapes: draw
    draw(c::Circle) = "drawing circle"
  end
end

using .Shapes.Polygons, .Shapes.Circles

# Deux méthodes de la même fonction: Shapes.draw
draw(Polygon())
draw(Circle())
```

6.2 Gestion des dépendances

Lors de l'utilisation d'un module son nom / chemin peut être spécifié de deux manières :

— chemin relatif (`using .ModName`) : dans ce cas, le code définissant le module est supposé avoir déjà été chargé. Avec un seul point (".") comme préfixe, le module à utiliser doit être défini dans la portée courante ; avec deux points (".."), il doit être défini dans la porté du module parent ; *etc.* (*cf.* exemples ci-dessus.)

— chemin absolu (`using ModName`) : dans ce cas, la commande `using`/`import` commence par charger la définition du module depuis un fichier source non encore évalué. La liste des modules utilisables de cette façon (ainsi que les chemins des fichiers sources à charger) est déterminée grâce à un ensemble de règles, qui guident donc la manière d'organiser le code source d'un projet conséquent.

6.2.1 Paquets et projets

Un "projet" Julia est un ensemble de fichiers organisés de la manière décrite au paragraphe 2.4.1 :

— `Project.toml` : la présence de ce fichier dans un répertoire signale que l'ensemble des fichiers du répertoire définit un "projet". Ce fichier contient aussi des méta-informations importantes sur le projet :

— un nom (ex: "ProjName"),

  - optionnellement, un identifiant (UUID [4], de la forme "26d708fe-1ec2-56ce-9e97-fa6abf607293") dont la présence transforme le "projet" en "paquet" (*i.e.* une bibliothèque fournissant un ensemble de fonctionnalités susceptible d'être utilisées dans un autre projet),
  - une liste de "paquets" externes que le projet peut utiliser comme dépendances,
  - un numéro de version, une liste d'auteurs, une licence logicielle et autres informations permettant de caractériser le projet ;
- `Manifest.toml` : ce fichier optionnel précise la version de chacune des dépendances listées dans `Project.toml` ;
- `src/ProjName.jl` : ce fichier source Julia définit un module de premier niveau appelé "ProjName" (du même nom que le projet lui-même), qui sera chargé par `using` ou `import` si le projet est utilisé comme dépendance d'un autre projet.

6.2.2 Environnements

Grâce aux informations contenues dans le fichier `Project.toml` (et éventuellement `Manifest.toml`), chaque projet définit un "environnement" : une liste de paquets dont le projet dépend.

À chaque instant, un (et un seul) environnement est "actif" : il fournit la liste des modules de premier niveau qu'il est possible d'utiliser à l'aide de `using` ou `import`. Lorsqu'un nouveau REPL est lancé, l'environnement actif par défaut est celui de Julia lui-même, qui porte le nom de la version courante (ex: "v1.1"). Le REPL possède un mode [5] spécifique permettant de gérer les environnements. On l'active en tapant "`]`", et il donne accès à de nombreuses commandes dont les deux principales sont :

- `activate` : pour activer l'environnement spécifique d'un projet ; le nom de l'environnement actif est affiché dans le prompt du REPL en mode "pkg".
- `add` : pour ajouter un paquet dans l'environnement actif (*i.e.* dans la liste des dépendances du projet en cours). Si le paquet est déjà installé sur le système, il s'agit juste d'ajouter la ligne correspondante dans `Project.toml`. Sinon, le paquet est au préalable téléchargé (depuis GitHub, en général) et installé.
- `instantiate` : installe toutes les dépendances de l'environnement actif. Ceci est particulièrement utile lorsqu'on travaille à plusieurs sur le même projet : à chaque fois que Bob récupère une version du projet modifiée par Alice, il peut ainsi installer les nouvelles dépendances qu'Alice a récemment ajoutées et que Bob n'a pas encore eu l'occasion d'utiliser.
- `test` : pour lancer la base de tests du paquet actif.

Command line

```
julia> # taper ] en début de ligne pour passer en mode "pkg"
(v1.1) pkg> activate .

(SandBox) pkg> add Lazy
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `.../SandBox/Project.toml`
  [50d2b5c4] + Lazy v0.13.2
  Updating `.../SandBox/Manifest.toml`
  [00ebfdb7] + CSTParser v0.5.2
  ...

(SandBox) pkg> test
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `.../SandBox/Project.toml`
 [no changes]
   Testing SandBox
 Resolving package versions...
Test Summary: | Pass  Total
SandBox       |   12     12
   Testing SandBox tests passed

(SandBox) pkg> # taper <backspace> pour quitter le mode "pkg"
julia>
```

4. Comme les paquets sont destinés à être publiés dans un écosystème plus large, cet identifiant permet de différencier deux paquets de noms identiques. À chaque création de projet, `Skeleton.jl` génère un UUID qui permettra de publier le projet sous forme de paquet par la suite si nécessaire

5. au même titre que le mode d'aide vu au début de ce document et activé en tapant "`?`" en début de ligne

Il est possible de lancer julia avec l'option "`--project=/path/to/project`" pour activer directement l'environnement d'un projet. Si le chemin du projet n'est pas défini ("`--project`"), Julia recherche le plus proche répertoire parent contenant un fichier `Project.toml` et active l'environnement correspondant.

En plus des paquets accessibles depuis l'environnement actif, il est toujours possible d'utiliser les paquets de la bibliothèque standard (par exemple : `Test`, `LinearAlgebra`, `Statistics`...). Il est en général aussi possible de charger les paquets installés dans l'environnement par défaut ("v1.1"). Ceci permet notamment :

— d'utiliser interactivement des paquets fournissant des outils de développement ou de débogage (`Revise`, `BenchmarkTools`, `Profile`...)
— de créer un script Julia "nu" (sans toute l'infrastructure liée aux projets) et d'y utiliser des paquets installés dans l'environnement par défaut.

NB : Dans un REPL en mode "pkg", la commande `test` lance la base de cas-tests en restreignant l'environnement aux seules dépendances du projet. Ceci permet de s'assurer que toutes les dépendances nécessaires ont bien été identifiées, et ainsi de garantir qu'un autre utilisateur installant le paquet dans son propre environnement n'aura pas de problème de dépendances.

6.3 Tests

Nous avons déjà rencontré quelques fonctionnalités du paquet `Tests`, qui fait partie de la bibliothèque standard Julia et fournit quelques outils utiles pour définir des cas-tests. Un exemple étant souvent plus parlant que de longues explications, voici une démonstration de son emploi :

test/runtests.jl

```
using Test

@testset "SandBox" begin
    @testset "addition" begin
        @test 1 + 1 == 2
    end

    @testset "multiplication" begin
        @test 1 * 1 == 1

        # Erreur !
        @test 2 * 1 == 1

        # Ce cas-test est connu pour être cassé
        @test_broken 2 * 2 == 5
    end
end
```

Command line

```
pkg> test
...
multiplication: Test Failed at /tmp/essai.jl:12
  Expression: 2 * 1 == 1
   Evaluated: 2 == 1
Stacktrace:
  ...
Test Summary:    | Pass  Fail  Broken  Total
SandBox          |    2     1       1      4
  addition       |    1                    1
  multiplication |    1     1       1      3
ERROR: LoadError: Some tests did not pass: 2 passed, 1 failed, 0 errored, 1 broken.
```

6.4 Documentation

6.4.1 Docstrings

Julia incorpore dans le langage la possibilité d'inclure la documentation des fonctions et types directement dans le code source. Ceci se fait, de manière assez similaire à Python, en faisant précéder la définition d'une fonction ou d'un

type par une chaîne de caractères (*docstring*) le documentant. Comme illustré ci-dessous, le contenu de la *docstring* peut être formaté en langage Markdown, pour une présentation plus riche.

```
"""
   my_gcd(a, b)

Calcule le PGCD des nombres `a` et `b`.

# Exemple
```julia-repl
julia> my_gcd(12, 18)
6
```
"""
function my_gcd(a, b)
  ...
end
```

Outre le fait que la proximité entre code et documentation permet plus facilement de garantir l'évolution conjointe des deux, l'intérêt d'une telle documentation est d'être accessible dans le REPL Julia (et les IDE qui en extraient l'information) :

Command line

```
help?> my_gcd
  my_gcd(a, b)

  Calcule le PGCD de a et b.

  Exemple
  =======

  julia> my_gcd(12, 18)
  6
```

Il est d'usage de ne documenter que les fonctions, mais pas chacune des méthodes individuelles. Ceci permet d'éviter d'être submergé par la documentation de 163 méthodes lorsqu'on demande de l'aide sur l'opérateur `+`. De ce fait, la documentation d'une fonction est centrée sur sa sémantique en général, plutôt que sur des informations concernant son implémentation spécifique pour un type de données particulier. Toutes les méthodes associées à une même fonction devraient se reconnaître dans la documentation de celle-ci.

6.4.2 Documenter.jl

Le paquet `Documenter.jl` s'impose comme la solution recommandée pour générer la documentation utilisateur d'un projet Julia. La documentation de `Documenter.jl` (elle-même réalisée à l'aide de ce paquet) est très complète, et nous ne mentionnons ici que brièvement les étapes qui restent à la charge de l'utilisateur lorsque le projet a été créé à l'aide de `Skeleton.jl` comme c'est le cas ici.

1. Modifier le fichier `docs/make.jl`. Pour un projet dont on cherche à construire la documentation HTML pour un usage interne, le fichier devrait ressembler à l'exemple ci-dessous.
2. Rédiger la documentation "hors code source" dans des fichiers markdown. Les fichiers de documentation et les titres associés sont listés dans le fichier `docs/make.jl`.
3. Si ce n'est déjà fait, ajouter des *docstrings* aux fonctions du projet (au moins l'API publique).
4. Générer la documentation en lançant le script `docs/make.jl`. Le résultat (par défaut au format HTML) est contenu dans le répertoire `docs/build`.

docs/make.jl

```
# Ajouter cette ligne au début
push!(LOAD_PATH,"../src/")

using Documenter, SandBox

makedocs(
    modules = [SandBox],
    format = :html,
    checkdocs = :exports,
    sitename = "SandBox.jl",

    # Compléter la liste des pages
    pages = Any["Accueil" => "index.md",
                "Titre 1" => "file1.md",
                "Titre 2" => "file2.md"]
)

# Commenter les lignes suivantes :
# deploydocs(
#     repo = "github.com/{GHUSER}/SandBox.jl.git",
# )
```

Command line

```
shell$ julia make.jl
[ Info: SetupBuildDirectory: setting up build directory.
[ Info: ExpandTemplates: expanding markdown templates.
[ Info: CrossReferences: building cross-references.
[ Info: CheckDocument: running document checks.
[ Info: Populate: populating indices.
[ Info: RenderDocument: rendering document.
[ Info: HTMLWriter: rendering HTML pages.
```