Zeste de savoir

À la découverte de Julia

12 août 2019

Table des matières

Dans cet article, je vais vous présenter un nouveau langage, Julia . Ce langage est en développement au MIT depuis 2009, et la première version publique date de 2012. Il est actuellement en phase de stabilisation des fonctionnalités pour sa version 0.4.

1. Un nouveau langage pour quoi faire ?

> Encore un nouveau langage !? Mais il n'y en a pas déjà assez ?
>
> *Un programmeur désabusé*

En effet, il existe déjà beaucoup — certains diront trop — de langages de programmation dans le monde. Il y a deux raisons principales à ce foisonnement :

- Beaucoup de langages ne sont utiles que dans un domaine spécifique : R pour les statistiques, SQL pour les bases de données relationnelles, …
- Dès qu'un groupe de *nerds* n'est pas satisfait des langages qui existent déjà, ils créent le leur. Cela ajoute d'autres langages génériques à la liste déjà longue (ce fut le cas du C, du C++, du Python à leur époque ; et de plein d'autre que le monde a oublié entre temps).

Dans le cadre de Julia, la création d'un nouveau langage est basée sur ces deux points à la fois. En effet, dans le domaine de la programmation scientifique (les auteurs de Julia parlent de *technical computing*), il existe plusieurs langages utilisables. Ces langages peuvent être classés en trois catégories :

Haut niveau	Bas niveau générique	Bas niveau spécialisé

Python	C	Assembleur
Matlab	C++	CUDA
Mathematica	FORTRAN	OpenCL
R	…	…
ACL		
Et tout un tas d'autres …		

Les langages de haut niveau sont souvent plus pratiques pour exprimer des idées et tester différentes méthodes. Les langages de bas niveau généraliste peuvent aussi être utilisés pour tester des méthodes, et offrent de meilleurs temps d'exécution (en général). Ils font payer cela par un temps de développement plus long (en général aussi, merci de ne pas ouvrir un troll ^^). Les langages de bas niveau spécialisé offrent des performances encore meilleures, contre un apprentissage complexe et un temps de débogage long.

En général, la création de logiciels scientifique suit le schéma suivant :

1. Écrire du code en Python (Matlab, …) pour pouvoir tester des idées et écrire plein de prototypes rapidement ;
2. Optimiser en implémentant les points bloquant (ou l'intégralité du programme) dans un langage plus bas niveau. Parce que même 5% de temps en moins sur 3 semaines, c'est quand même bien.
3. *Si besoin — et si l'on sait faire — utiliser un ou plusieurs langages spécifiques pour améliorer encore les performances. Il est aussi possible d'utiliser des outils comme MPI ou OpenMP pour paralléliser le code.*

C'est ce que l'on appelle le **problème des deux langages** : on est obligé d'utiliser deux (ou plus) langages différents pour avoir à la fois de bonnes performances et exprimer facilement ses idées. C'est un problème parce que les gens qui font de la programmation technique ne sont généralement pas intéressés par le code : ils veulent utiliser l'informatique comme un outil, pas passer des heures à programmer.

Et c'est à ce problème spécifique que s'attaque Julia. Dans l'article originel de présentation de Julia, les auteurs affirment que :

> Julia has the performance of a statically compiled language while providing interactive dynamic behavior and productivity like Python, LISP or Ruby.
>
> *Bezanson, J., Karpinski, S., Shah, V. B., & Edelman, A. (2012). Julia : A fast dynamic language for technical computing.* ↗

2. Comment ça marche ?

2.1. Un brin de magie

La technologie magique[1] qui permet ce *quasi-miracle* de performances et d'interactivité est celle de compilation à la volée, ou de compilation *Just In Time* (JIT) en anglais.

Cela consiste principalement à compiler à la volée le code source en code machine. À chaque fois que l'utilisateur entre du code dans l'interpréteur, ce dernier est compilé et exécuté sous forme de code machine.

Cette compilation JIT est effectuée en utilisant le projet LLVM , qui consiste en un ensemble d'outils pour créer des compilateurs.

2.2. Et des bibliothèques

Julia utilise aussi pas mal d'autres bibliothèques compilées pour certaines de ses fonctionnalités :

- `libuv` (la bibliothèque derrière Node.js) pour la gestion des entrées et sorties de manière asynchrone ;
- `BLAS` et `LAPACK` pour l'algèbre linéaire ;
- `FFTW` pour les transformées de Fourier ;
- `libmojibake` pour la gestion de l'Unicode ;
- Et quelques autres bibliothèques maison :
  - `openlibm` (une implémentation générique de la bibliothèque mathématique C) ;
  - `openspecfun` pour les fonctions spéciales ;
  - et `libosxunwind` pour les stacktraces sous OS X.

Le cœur du langage est écrit en C, et le parseur en LISP. La quasi-intégralité de la bibliothèque standard est écrite directement en Julia.

Tout ça pour dire que même si on a là un nouveau langage, les auteurs ne réinventent pas non plus la roue.

3. Bon, et ça ressemble à quoi ?

i

Vous pouvez tester tout le code de cet article depuis votre navigateur, soit avec un notebook temporaire (le choix du langage est à droite), soit sur JuliaBox si vous avez un compte Google — vous pouvez y utiliser un notebook, et même lancer une console pour les plus barbus d'entre vous. L'interpréteur se lance avec la commande `julia`.

1. "Toute technologie suffisamment avancée est indiscernable de la magie." (*Arthur C. Clarke*)

3.1. Une syntaxe simple

La syntaxe de Julia est plutôt simple. Elle ressemble beaucoup à celle de Matlab, ou de Ruby. Un exemple ? Voyez-vous même :

```
a = 5
# a est un entier
b = 67.67e42
# b est un flottant

# La variable a est ré-affectée dynamiquement au type String
a = "Bonjour"

c = a + 3 * b^4

d = sin(atan(3)) # Utilisation de fonctions
```

Ce qui est bien, c'est que comme on est en 2015, les variables Unicode sont autorisées :

```
α = 56

ΔΨ = 78 * α
```

On les tape depuis la console `julia`, qui connaît un certain nombre de complétions LaTeX : `\alpha` + tab donne .

Les chaînes de caractères aussi, et on peut même faire de l'interpolation de chaînes à la mode Perl :

```
toi = "K¬öba†‹f›"
println("Bonjour, $toi !") # println affiche une ligne de texte
println("12 est de type $(typeof(12))")
```

Le typage de Julia est **fort**, **dynamique** et **inféré**. Le compilateur devine automatiquement le type des variables :

```
a = 12   # Int64 ou Int32 selon les machines
b = 12.0 # Float64
c = "Bonjour, ça zeste ?" # UTF8String
d = true # Bool
e = [12, 34, 45] # Array{Int64}
f = [12.0, 34, "Youhou !"] # Array{Any}
```

3. Bon, et ça ressemble à quoi ?

Julia est un langage qui a ses origines à la fois dans la programmation orientée objet, et dans la programmation fonctionnelle. Il est possible (et même obligatoire) de définir des fonctions :

```
# Déclaration de fonction sans annotations de type
function add(a, b)
    return a + b
end

# Déclaration de fonction avec annotations de types
function add(a::String, b::String)
    "$a & $b" # la dernière instruction est la valeur de retour
end

# Déclaration de fonctions sous forme courte :
f(x) = 42 * x^3

# Et même de lambdas :
g = x -> 42 / x^6
```

Avec Julia, tous les objets sont des citoyens de première classe : les types utilisateurs sont aussi puissants, compacts et rapides que les types de base du langage. Il existe trois versions de types : les types abstraits, les types normaux et les types immuables (non modifiable) :

```
abstract Vehicule

type Voiture <: Vehicule
    modele::String       # Variable membre de type String
    vitesse_max::Float64 # Variable membre de type Float64
end

immutable Velo <: Vehicule
    pignons::Integer
end

# On crée les objets avec la syntaxe suivante
velo1 = Velo(12) # un vélo à 12 pignons
velo2 = Velo(15) # un vélo à 15 pignons

println(velo1.pignons) # Les attributs sont publiques
println(velo2.pignons)
```

L'opérateur `<:` est l'opérateur d'héritage `est un`. On ne peut créer de types dérivés (des types hérités) que depuis les types abstraits, et il est impossible de créer un objet ayant un type abstrait. Les types peuvent aussi être paramétrés (on retrouve là des outils de programmation générique à la C++) par d'autres types, ou par des objets immuables :

```julia
# Type Vendeur paramétré par un type héritant de véhicule
type Vendeur{T<:Vehicule}
    stock::Integer
    véhicules::Vector{T} # Tableau à une dimension (Vector) de T
end

# On peut avoir un objet vendeur de vélos
boutique_de _velos = Vendeur{Velo}(2, [velo1, velo2])
# Et un concessionnaire automobile
concessionnaire = Vendeur{Voiture}(0, Voiture[])

# Le paramétrage peut aussi être déterminé automatiquement par le
    compilateur
vendeur_velos = Vendeur(1, [vélo1])

println(typeof(vendeur_velos))
# -> Vendeur{Velo}
```

3.2. Un objet, des méthodes : le dispatch multiple

Julia est orienté objet, mais avec un modèle qui n'est pas le même que celui de Python, Java ou C++. Ici, il n'y a aucune fonction membre (aussi parfois appelées méthodes) définie à l'intérieur des types. À la place, Julia utilise le concept de *dispatch multiple* pour implémenter son modèle objet. L'idée est de sélectionner une version spécifique d'une fonction en fonction de l'ensemble des types des paramètres.

Prenons un exemple : il existe plusieurs types de véhicules. Parmi eux, des véhicules à roues et des véhicules sans roues. Parmi les véhicules à roue, on peut retrouver les vélos, les motos et les voitures.

FIGURE 3. – Relations entre objets en Julia

Sur des objets de type véhicule, plusieurs types de **fonctions** peuvent être utilisés : avancer pour tous les véhicules, rouler pour les véhicules à roues, remplir le réservoir pour les véhicules à essence, … Mais la manière de le faire dépendra partiellement du type d'objet considéré. Chaque fonction sera donc implémentée (spécialisée) pour les types d'objets, chaque implémentation étant appelée une **méthode** dans le monde de Julia.

Dans certains cas, il n'est pas nécessaire de spécialiser explicitement une fonction, le compilateur pouvant se charger de spécialiser un algorithme pour les différents arguments. C'est ainsi que toutes les opérations sur les matrices sont implémentées, quel que que soit le types de matrice : Triangulaire, Hermitienne, … Seules certaines fonctions sont spécialisées pour prendre en compte les spécificités (typiquement le calcul des valeurs propres).

Voici comment on pourrait implémenter notre exemple de véhicules :

```
abstract Vehicule
abstract VehiculeARoues <: Vehicule
abstract VehiculeARouesEtAEssence <: VehiculeARoues

type Bateau <: Vehicule end
type Velo <: VehiculeARoues end
type Voiture <: VehiculeARouesEtAEssence end
type Moto <: VehiculeARouesEtAEssence end

# Définition des fonctions : une version générique
function avancer(v::Vehicule)
    println("En avant !")
end

function avancer(v::VehiculeARoues)
    # pour les véhicules à roues, la fonction avancer appelle
    # directement la fonction rouler
    rouler(v)
end

function rouler(v::VehiculeARoues)
    println("Ça roule !")
end

function remplir_le_reservoir(v::VehiculeARouesEtAEssence)
    println("Glou glou glou …")
end

bateau = Bateau()
velo = Velo()
voiture = Voiture()
moto = Moto()

# La détermination de la bonne méthode est faite automatiquement:
avancer(bateau)   # -> En avant !
avancer(velo)     # -> Ça roule !
avancer(voiture) # -> Ça roule !
avancer(moto)     # -> Ça roule !
```

```
39
40 remplir_le_reservoir(voiture) # -> Glou glou glou …
41
42
43 # S'il n'existe pas de méthode adaptée, une erreur est levée
44 remplir_le_reservoir(velo)
45 # ERROR: MethodError: `remplir_le_reservoir`
46 # has no method matching remplir_le_reservoir(::Velo)
47
48 # On peut encore spécialiser les méthodes :
49 function rouler(m::Moto)
50     println("Broouum !")
51 end
52
53 # La fonction générique est appelée
54 avancer(voiture) # -> Ça roule !
55 # La fonction spécialisée est appelée
56 avancer(moto)    # -> Broouum !
```

Les différentes méthodes peuvent être spécialisées manuellement ou automatiquement par le compilateur. Il n'est jamais nécessaire de préciser les types, et la fonction la plus spécialisée sera toujours appelée lors de l'exécution. Et cette spécialisation a lieu sur les types de l'ensemble des paramètres :

```
1  # Fonction par défaut, sans spécification de types
2  function foo(a, b) # version 1
3      # Cette fonction peut être utilisée pour tous les types
4      # qui implémentent l'opérateur +
5      return a + b
6  end
7
8  function foo(a::Integer, b) # version 2
9      println("Le premier argument est un entier")
10     return a + b
11 end
12
13 function foo(a, b::Integer) # version 3
14     println("Le second argument est un entier")
15     return a + b
16 end
17
18 function foo(a::Integer, b::Integer) # version 4
19     println("Les deux arguments sont des entier")
20     return a + b
21 end
22
23 function foo(a::String, b::Integer) # version 5
24     println("Le premier argument est une chaine, ",
```

```
            "et le second argument est un entier")
    return string(a, b)
end

foo(3.6, 5.9)      # utilise la version 1
foo(3, 5.9)        # utilise la version 2
foo(3.6, 6)        # utilise la version 3
foo(4, 2)          # utilise la version 4
foo("Bonjour ", 5) # utilise la version 5

# Comme aucune autre version n'est définie, cet appel utilise la
    version 1
# Mais ceci renvoie une erreur, car il n'y a pas d'opération +
    entre
# chaînes.
foo("Bonjour ", "le monde")

function foo(a::String, b::String)
    return a * b # La concaténation se fait avec l'opérateur *
end

# On a défini une fonction entre temps, tout va bien.
foo("Bonjour ", "le monde")
```

Ce fonctionnement basé sur le *multiple dispatch* est entre autres ce qui permet à Julia d'obtenir sa rapidité à l'exécution. En effet, les fonctions sont compilées à la volée pour chaque jeu d'argument, ce qui permet à chaque fois d'avoir du code natif optimisé. On peut explorer ce code avec la fonction `code_native`, qui affiche l'assembleur obtenu (`julia>` est le prompt de l'interpréteur) :

```
julia> function bar(a, b) return a + b end
bar (generic function with 1 method)

julia> code_native(bar, (Float64, Float64))
        .section        __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 1
        push        RBP
        mov         RBP, RSP
Source line: 1
        vaddsd      XMM0, XMM0, XMM1
        pop         RBP
        ret

julia> code_native(bar, (Int, Int))
        .section        __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 1
```

```
        push        RBP
        mov         RBP, RSP
Source line: 1
        add         RDI, RSI
        mov         RAX, RDI
        pop         RBP
        ret
```

Même si comme moi vous n'y connaissez rien en assembleur, vous remarquerez que les instructions sont spécialisées à la fois pour l'architecture du processeur, et pour les types utilisés. Cette spécialisation a aussi lieu pour les types définis par les utilisateurs.

```
julia>  immutable Point
            x::Float64
            y::Float64
        end

julia>  function add(p::Point, q::Point)
            return Point(p.x + q.x, p.y + q.y)
        end
add (generic function with 1 method)

julia> # Le code LLVM est quand même plus lisible que l'assembleur

julia> @code_llvm add(Point(3.4, 4.4), Point(3.5, 6.5))

define %Point @julia_add_42523(%Point, %Point) {
top:
  # Addition des valeurs de x
  %2 = extractvalue %Point %0, 0, !dbg !504
  %3 = extractvalue %Point %1, 0, !dbg !504
  %4 = fadd double %2, %3, !dbg !504
  # Insertion du résultat de l'addition dans un nouveau point
  %5 = insertvalue %Point undef, double %4, 0, !dbg !504
  # Addition des valeurs de y
  %6 = extractvalue %Point %0, 1, !dbg !504
  %7 = extractvalue %Point %1, 1, !dbg !504
  %8 = fadd double %6, %7, !dbg !504
  # Insertion du résultat de l'addition dans le nouveau point
  %9 = insertvalue %Point %5, double %8, 1, !dbg !504, !julia_type
      !506
  # Et on retourne le nouveau point
  ret %Point %9, !dbg !504
}

# À comparer au code créé pour la même fonction définie sur des
    tuples:
julia> function add(p::(Float64,Float64), q::(Float64,Float64))
```

```
            return (p[1]+q[1], p[2]+q[2])
        end
add (generic function with 2 methods)

julia> @code_llvm add((3.4, 4.4), (3.5, 6.5))

define <2 x double> @julia_add_42670(<2 x double>, <2 x double>) {
top:
  %2 = extractelement <2 x double> %0, i32 0, !dbg !965
  %3 = extractelement <2 x double> %1, i32 0, !dbg !965
  %4 = fadd double %2, %3, !dbg !965
  %5 = insertelement <2 x double> undef, double %4, i32 0, !dbg
      !965, !julia_type !967
  %6 = extractelement <2 x double> %0, i32 1, !dbg !965
  %7 = extractelement <2 x double> %1, i32 1, !dbg !965
  %8 = fadd double %6, %7, !dbg !965
  %9 = insertelement <2 x double> %5, double %8, i32 1, !dbg !965,
      !julia_type !967
  ret <2 x double> %9, !dbg !965
}
```

3.3. Autres points intéressants

3.3.1. Interface vers le C ou le Fortran

Comme on n'allait pas se passer de tout un tas de bibliothèques juste pour le plaisir, il est extrêmement facile d'appeler du C ou du Fortran depuis Julia (l'intégration du C++ est en cours de développement). Ce mécanisme est appelé FFI (*Foreign function interface*). Par exemple, si vous avez un fichier C qui contient la fonction suivante

```c
/* Add 'a' to all the values in the array 'b' of size 'n',
 * and return the mean value of 'a'.
 */
double foo(int a, int* b, int n){
    int i = 0;
    double res = 0;
    for (i=0; i<n; i++){
        b[i] += a;
    }

    for (i=0; i<n; i++){
        res += b[i];
    }

    return res/n;
}
```

Et que vous le compilez sous la forme d'une bibliothèque partagée `libbar.so` ou `bar.dll`, vous pouvez alors appeler la fonction `foo` depuis Julia aussi simplement que ça :

```
a = 5
b = [4, 5, 6, 7]
c = ccall((:foo, :bar), Float64, (Int32, Ptr{Int32}, Int32), a, b,
    length(b))
```

Il y a correspondance exacte en mémoire des types C et Julia, et la conversion est faite automatiquement par la fonction `ccall`. L'utilisation de code Fortran, se fait exactement de la même manière.

3.3.2. Les macros : du code qui créé du code

Une autre capacité intéressante de Julia réside dans sa capacité à utiliser des macros, dans l'esprit de Lisp. Une macro est un bout de code qui est capable de créer d'autre code à l'exécution. Par exemple, la fonction `printf` du C est implémentée en Julia sous la forme d'une macro : `@printf`. Cette macro génère donc du code spécialisé pour chaque invocation, code qui sera compilé à chaque fois.

Dans l'exemple qui suit, du code spécialisé est généré pour afficher une chaîne de caractères, et un entier. La fonction `macroexpand` force la génération des macros, et affiche le code correspondant.

```
macroexpand(:(@printf("Test: %s ", "Brocolis")))
# quote
#     #72#out = Base.Printf.STDOUT
#     #73###x#2591 = "Brocolis"
#     local #66#neg, #67#pt, #68#len, #69#exp, #70#do_out, #71#args
#     Base.Printf.write(#72#out,"Test: ")
#     begin
#         Base.Printf.print(#72#out,#73###x#2591)
#     end
#     Base.Printf.write(#72#out,' ')
#     Base.Printf.nothing
# end

macroexpand(:(@printf("Test: %d", 42)))
# quote
#     #75#out = Base.Printf.STDOUT
#     #76###x#2592 = 42
#     local #81#neg, #80#pt, #79#len, #74#exp, #77#do_out, #78#args
#     Base.Printf.write(#75#out,"Test: ")
#     if Base.Printf.isfinite(#76###x#2592)
#         (#77#do_out,#78#args) =
    Base.Printf.decode_dec(#75#out,#76###x#2592,"",0,-1,'d')
```

```
#         if #77#do_out
#             (#79#len,#80#pt,#81#neg) = #78#args
#             #81#neg && Base.Printf.write(#75#out,'-')
#
    #Base.Printf.write(#75#out,Base.Printf.pointer(Base.Printf.DIGITS),#80#pt)
#         end
#     else
#         Base.Printf.write(#75#out,begin  # printf.jl, line 143:
#                 if Base.Printf.isnan(#76###x#2592)
#                     "NaN"
#                 else
#                     if (#76###x#2592 #Base.Printf.< 0)
#                         "-Inf"
#                     else
#                         "Inf"
#                     end
#                 end
#             end)
#     end
#     Base.Printf.nothing
# end
```

Du code différent est généré pour chaque appel à la macro, code qui pourra être compilé et être optimisé pour chaque appel à la macro.

3.4. Et beaucoup d'autres choses !

Je n'ai pas le temps de parler de tout, mais Julia a encore plein de fonctionnalités sympas :

- La programmation parallèle en mémoire distribuée est intégrée au langage (la mémoire partagée est en cours de développement) ;
- Il existe des packages pour appeler du code Python, Java, R, et donc utiliser les immenses bibliothèques disponibles dans tous ces langages ;
- Des fonctionnalités d'interpréteur de commande shell ;
- Et une communauté chaleureuse et accueillante qui créé plein de choses avec ce nouveau langage !

4. Et c'est performant ?

> Oui, parce que c'est bien beau d'avoir un language tout neuf, mais s'il ne crache pas des gigaflops, ce n'est pas très intéréssant.
>
> *Notre programmeur désabusé*

Eh bien oui, le langage se défend pas mal ! C'est avant tout un langage dynamique où il est possible d'écrire des boucles comme en C, et d'avoir les performances du C. Il y a un premier jeu de benchmark sur le site de Julia, mais j'aimerai vous présenter un autre graphique :

Source : La liste de diffusion julia-user

Ce graphique compare la taille des codes sources, et la rapidité d'exécution sur un ensemble d'algorithmes. Et il est clair que Julia se situe dans le bon coin : celui des bonnes performances, et d'une faible taille de code, donc d'une bonne expressivité.

Alors certes il ne s'agit que de benchmarks, certes ce ne sont que des exemples simples, toutefois le résultat est impressionnant.

Pour un exemple de code plus conséquent, cette discussion concerne l'implémentation des bibliothèques de FFT en pur Julia. Et les performances actuelles sont exactement comparables à celles de FFTPACK ou de FFTW, tout en ayant seulement 1/3 du nombre de lignes !

5. Mais on peut s'en servir en vrai ?

Selon votre domaine, Julia sera plus ou moins facilement utilisable. Ainsi, pour faire des sites web, peu de frameworks existent, et il vous faudra écrire beaucoup de code bas niveau. De même, pour faire des statistiques, même si beaucoup de packages existent, Julia est encore loin derrière R. Par contre, pour faire de l'optimisation mathématique, de la manipulation d'images ou du *machine learning*, vous trouverez tout ce qu'il vous faut.

D'autre part, le langage est encore en pleine évolution. Si la série de versions 0.3 à 0.3.7 sont toutes compatibles, la version 0.4 introduit de gros changements non rétro-compatibles. En pratique , sur les 8 mois que j'ai passés à m'amuser avec ce langage, je n'ai eu besoin de mettre à jour mon code qu'une seule fois à cause des changements introduits.

Dans tous les cas, les versions stables sont relativement stables, et le langage est mature et utilisable. Il est déjà utilisé en production pour de l'analyse de donnée à grande échelle, et quelques articles scientifiques l'utilisant commencent à apparaitre.

6. En savoir plus

J'espère vous avoir donné envie de tester Julia, pour l'installation sur votre machine, c'est par là ! Voici quelques liens pour vous accompagner dans votre apprentissage, et n'hésitez pas à utiliser le forum, je serais ravi de vous aider !

— Apprendre les bases de Julia en 15 min
— Apprendre Julia par l'exemple
— Quelques trucs pour faire la transition depuis le C/C++/Python/...

— La documentation du langage , très bien écrite. C'est la référence une fois passé les trois liens ci-dessus.
— La liste de diffusion . C'est là qu'il faut venir poser ses questions pour avoir une réponse par les développeurs principaux.
— Why we created Julia : des explications sur le design du langage

Et quelques liens plus techniques :

— Un notebook à propos du dispatch multiple
— Un article scientifique par les auteurs de Julia

Merci à Kje pour m'avoir incité à me lancer dans cet article et pour l'avoir relu dans tous les sens, et à toute la communauté de ZdS pour donner envie d'écrire ici !

Liste des abréviations

JIT Just In Time. 3