eBook Gratuit

APPRENEZ wolfram-mathematica

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

#wolfram-mathematic

a

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: wolfram-mathematica

It is an unofficial and free wolfram-mathematica 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 wolfram-mathematica.

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 wolfram-mathematica

Remarques

Cette section donne un aperçu de ce qu'est wolfram-mathematica et pourquoi un développeur peut vouloir l'utiliser.

Il devrait également mentionner tous les grands sujets de wolfram-mathematica, et les relier aux sujets connexes. La documentation de wolfram-mathematica étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Examples

Qu'est-ce que (Wolfram) Mathematica?

Wolfram définit Mathematica comme "le système définitif du monde pour l'informatique technique moderne". Une déclaration audacieuse qui est partiellement vraie. Ce n'est probablement pas le système le plus prédominant (car vous devez payer un peu pour une utilisation commerciale), les utilisateurs utilisent par exemple Python ou R. Qu'est-ce que c'est, est l' **environnement** le plus complet pour "l'informatique technique" en fournissant les fonctionnalités suivantes:

- Le langage Wolfram: un langage multi-paradigme qui couvre le calcul symbolique avec la programmation procédurale, fonctionnelle, basée sur les listes et les règles
- "Notebooks": une combinaison de documentation, de programmes et de résultats
- Wolfram Algorithmbase: Probablement le plus grand ensemble d’algorithmes organisés couvrant la plupart des domaines mathématiques, informatiques et graphiques. La plupart des gens ne font pas la distinction entre Algorithmbase et le langage car ils sont étroitement liés
- Avec l'introduction de Wolfram | Alpha, Wolfram Knowledgebase couvre de nombreux domaines de connaissances courants, de sorte que vous pouvez répondre à une question comme "temps total pour décongeler une dinde de 10 livres dans de l'eau froide" (5h) dans votre programme.

L'ensemble du système s'exécute sur le «moteur Wolfram», qui est essentiellement une machine virtuelle, un peu comme la machine virtuelle Java ou le Common Language Runtime de Microsoft permettant l'exécution sur un ensemble diversifié de plates-formes Windows, Mac et Linux. En plus des programmes exécutés sur un ordinateur, vous pouvez également les exécuter dans le "Wolfram Cloud", un processus simple par rapport à d'autres langages de "niveau inférieur" tels que Java et C #.

La version actuelle de Mathematica 11 peut être exécutée sur le bureau, Wolfram Cloud et iOS (iPad et iPhone).

Lire Démarrer avec wolfram-mathematica en ligne: https://riptutorial.com/fr/wolfram-mathematica/topic/3004/demarrer-avec-wolfram-mathematica

Chapitre 2: Ordre d'évaluation

Remarques

Ceci est censé expliquer l'ordre d'évaluation aussi clairement que possible. Il est probablement moins idéal comme introduction à l'exécution de Mathematica. Il se base sur le `tutorial/Evaluation` page d' `tutorial/Evaluation` de Mathematica en expliquant l'ordre dans lequel différentes règles sont appliquées et en expliquant quelles fonctions sont traitées spécialement par le moteur d'évaluation.

Contextes d'évaluation

A tout moment, le code s'exécute dans un contexte. Un contexte consiste en un ensemble de règles de remplacement (son dictionnaire) et une priorité dans laquelle elles doivent être évaluées. Nous regroupons cinq types de contextes avec un comportement et des restrictions distincts: * Définir le contexte * Contexte du bloc * Contexte adapté * Remplacer tout le contexte * Remplacer le contexte répété

Définir les contextes

Un ensemble de contextes est celui défini par un ensemble d'opérations Set [] (la fonction la plus couramment écrite `=`). L'exemple principal est l'environnement d'exécution principal. Son dictionnaire est modifié chaque fois qu'un ensemble [] est exécuté sur une variable qui n'est pas autrement étendue. Les autres instances de Set Contexts proviennent de packages. Un contexte de package est lié à son contexte parent en ce sens que tous les motifs du package deviennent également des motifs dans le contexte parent, avec un préfixe approprié (une définition `foo[x_]=3*x` devient `InnerPackageName\` foo [x _] = 3 * x `).

Set Contexts ne peut contenir que des règles associées à une "Tag", une chaîne associée à la tête pour identifier plus rapidement l'applicabilité de la règle. Une application de `Set[yyy_,zzz_]` déterminera une étiquette en vérifiant si `yyy` est un symbole. Si c'est le cas, alors c'est l'étiquette. Sinon, il vérifie `yyy[[0]]` , puis `yyy[[0,0]]` , etc. Si, à un moment donné, il s'agit d'un symbole, alors cela est considéré comme l'étiquette. S'il s'agit d'un atome autre qu'un symbole (tel que String, Integer, Real ...), une erreur est générée et aucune règle ne sera créée.

Les fonctions UpSet (écrit `^=`) et TagSet (écrit `/:` / `=`) (et leurs cousins UpSetDelayed et TagSetDelayed) permettent d'associer une règle à une autre étiquette. Il y a toujours la restriction que l'étiquette doit être un symbole. UpSet l'associera à chacun des arguments de l'expression, ou à leur tête s'ils sont une fonction avec un symbole pour une tête. Par exemple, appeler UpSet sur `f[a,b,c+d,e[f,g],5,h[i][j][k],p_]` associera la règle créée avec `a` , `b` , `e` , et `h` . Les arguments `c+d` , `5` et `p_` ne seront associés à rien et provoqueront l'affichage d'un message d'erreur. L'affectation réussira toujours sur chacun des arguments et (comme cela sera précisé plus loin dans l'ordre d'évaluation), elle fonctionnera toujours dans presque tous les cas. TagSet est comme UpSet, mais vous pouvez spécifier exactement un symbole pour la balise. Le symbole doit

toujours être quelque chose qui pourrait être défini par Set ou UpSet (un symbole de niveau supérieur dans la tête ou les arguments). Par exemple, `TagSet[f, f[a,b[c]], 2]` est acceptable et associera la définition à `f` ; `TagSet[a, f[a,b[c]], 2]` et `TagSet[b, f[a,b[c]], 2]` sont également acceptables, mais `TagSet[c, f[a,b[c]], 2]` ne l'est pas.

Les règles à l'intérieur d'un contexte nécessitent une priorité d'application, car il peut exister de nombreuses règles qui s'appliquent à une expression donnée. (Cela est également vrai dans les contextes ReplaceAll et ReplaceRepeated, mais ils le font très différemment). La priorité est généralement destinée à correspondre à la *spécificité* du motif. Étant donné une expression `a[q][b[c,d],e[f,g]]` pour évaluer, avec la tête et les arguments tous évalués aussi complètement qu'ils le seront (voir ci-dessous), commencez par chercher les règles marquées avec `b` qui s'applique; puis les règles étiquetées avec `e` ; puis les règles étiquetées avec `a` . Dans chaque ensemble de règles, un ordre est maintenu pour ceux associés à un symbole donné. Les règles sans espace (telles que `f[a,b]=3`) sont automatiquement placées en haut et triées dans l'ordre canonique (l'ordre de Sort). Chaque fois qu'une nouvelle règle est ajoutée, le noyau passe par la liste; Si certaines règles ont exactement le même LHS, elles sont remplacées sur place. Sinon, il effectue une comparaison de spécificité. Si une règle X déjà dans la liste est déterminée comme étant "moins spécifique" que la nouvelle règle Y, alors Y est placé immédiatement avant X. Sinon, elle continue dans la liste. Si aucune règle n'est moins spécifique, la règle est placée à la fin de la liste. Le contrôle de spécificité est plus compliqué et plus détaillé dans la section ci-dessous.

Spécificité de la règle

* Si deux expressions ont aucune instance de BlankSequence (__), BlankNullSequence (___), en option (:), Alternatives (|), répétée (..), RepeatedNull (...), ou des arguments optionnels (_.), Puis ils peuvent être comparés structurellement. Étant donné deux arbres d'expression équivalents X et Y, où tous les espaces dans Y sont également des blancs dans X, mais X comporte des espaces vides où Y ne l'est pas, alors Y est plus spécifique. * Si deux expressions sont équivalentes, sauf que certaines instances de _ ont été remplacées par __ dans l'autre expression, ou que __ ont été remplacées par ___ , alors la première est plus spécifique. * Si un décapage plus en option (:) (ou en option _.) Termes donne l'autre expression, alors celle - ci est plus spécifique. * Si un certain ensemble de choix parmi Alternatives donne l'autre expression, alors celle-ci est plus spécifique. * Si remplacer toutes les instances de `RepeatedNull[foo]` par `Repeated[foo]` , ou `Repeated[foo]` par `foo` , donne l'autre expression alors cette dernière est plus spécifique * Certaines combinaisons de ces règles peuvent être appliquées en même temps, mais ce n'est pas le cas savoir ce que sont les cas pour cela. * Les combinaisons d'expression telles que `_List` et `{___}` devraient théoriquement les traiter de manière identique, mais la comparaison semble étrangement dépendante du contexte, les classant parfois d'une manière ou d'une autre.

Contexte du bloc

Un contexte de bloc est plus restrictif, car le LHS d'une règle dans un bloc ne peut être qu'un symbole. C'est-à-dire que seules les définitions de la forme `f=2+x` , pas de `f[x_]=2+x` . (Notez que, d'un point de vue pratique, les fonctions peuvent toujours être construites avec des définitions telles que `Set [Block est lié à son contexte parent en ce sens que toute nouvelle définition lors de l'évaluation du Block est transmise normalement va "ombrer" un ensemble de variables,

fournissant des définitions qui peuvent masquer celles du contexte environnant. Les définitions du contexte environnant sont toujours accessibles pendant l'évaluation de l'expression interne. Comme il ne peut y avoir qu'une définition associée à un symbole, il n'y a aucune notion de priorité comme ci-dessus.

Contexte assorti

Après la correspondance d'une règle, il existe des définitions liées localement pour les variables. Cela se produit *lexicalement* . C'est-à-dire qu'il se substitue aux définitions liées aux variables de l'expression, sans rien évaluer d'autre. Ce n'est qu'une fois que toutes les souscriptions ont eu lieu qu'elle recommence à évaluer l'expression dans son ensemble. La principale méthode de création des contextes appariés est une règle d'un contexte ou d'une règle d'ensemble. Par exemple, dans

```
g[a_]:=a+x;
f[x_]:=x+g[1];
f[x^2]
(*Yields 1+x+x^2 *)
```

En appariant la règle `f[x_]` à `f[y]` , le symbole `x` est lié à la valeur `x^2` . Il effectue la substitution unique, mais comme il n'évalue pas `g` , il renvoie `x^2+g[1]` . Ceci est ensuite évalué à nouveau dans le contexte entourant et devient `1+x+x^2` . La différence significative dans l'évaluation dans un contexte apparié est que le remplacement n'est *pas* récursif. Quand il supplante `x->x^2` , il ne répète pas même sur ses propres résultats.

Les contextes appariés sont également créés par `With` , `Module` , `Function` et `Replace` notamment. De nombreuses autres fonctions les créent en interne. Par exemple, `Plot` utilise ce type de contexte pour évaluer l'expression à tracer.

Remplacer le contexte répété

Un contexte ReplaceRepeated est créé lorsqu'une application de `ReplaceRepeated` se produit. Ceci est distinct en ce sens qu'il peut avoir n'importe quelle expression en règle générale, y compris celles sans balise, telles que `_[_]` . En ce sens, c'est le contexte le plus flexible. Il peut également inclure plusieurs règles pouvant entrer en conflit, il doit donc conserver une priorité. Un contexte ReplaceRepeated appliquera d'abord la première règle de la liste, le cas échéant. S'il ne correspond pas, il passe à la seconde, et ainsi de suite. Si à tout moment une règle correspond, elle retourne à la première règle et recommence. Si, à tout moment, une application de règle se produit et qu'aucune modification ne se produit, elle se terminera - même si d'autres règles plus loin dans la liste apporteraient des modifications. Cela signifie que toute règle moins spécifique précédemment dans la liste empêchera les règles ultérieures d'être utilisées. De plus, placer `a_->a` en tête de la liste des règles entraînera la fin immédiate de toute la fonction `ReplaceRepeated` .

ReplaceAll Context

Un contexte ReplaceAll est créé lorsqu'une application de `ReplaceAll` se produit. Son fonctionnement est similaire à celui de ReplaceRepeated en ce sens que la priorité de son

application de règle est dans la liste lorsque deux peuvent s'appliquer au même niveau de l'expression. Cependant, c'est comme un contexte apparié en ce sens que les contenus remplacés ne sont pas évalués plus loin, *même par des règles ultérieures* . Par exemple, `x/.{x->y,y->z}` produit `y` . Il est donc incorrect de voir une application de ReplaceAll appliquer chaque règle à son tour. Au lieu de cela, il traverse l'arbre, à la recherche de règles applicables. Lorsqu'il trouve quelque chose qui correspond, il exécute le remplacement, puis retourne l'arborescence sans parcourir le nouvel arbre. Il convient également de noter qu'elle tente d'appliquer des règles de haut en bas, ce qui peut entraîner un dérèglement de la liste. Par exemple,

```
Cos[1 + 2 Sqrt[Sin[x]]] /. {Cos[_] -> 5, Sin[_] :> (Print[1]; 10)}
Cos[1 + 2 Sqrt[Sin[x]]] /. {Sin[_] :> (Print[1]; 10), Cos[_] -> 5}
```

les deux donnent `5` sans rien imprimer. Parce que le `Cos[_]` correspond à un niveau supérieur de l'arborescence, celui-ci est appliqué en premier.

`Hold` et `Evaluate` et l'ordre d'exécution

L'ordre d'évaluation, sous la forme d'une expression, procède comme suit: * Évaluez la tête aussi complètement que possible * Si la tête a une propriété `Hold` (`HoldFirst` , `HoldRest` , `HoldAll` , `HoldAllComplete`), alors * Vérifiez les arguments pertinents. À moins que ce soit `HoldAllComplete` , vérifiez si le responsable est `Evaluate` . Si c'est le cas, alors supprimez le `Evaluate` et marquez-le pour qu'il soit évalué de toute façon. * Vérifiez les arguments des instances de `Unevaluated` et `Unevaluated` -les si nécessaire, à moins que la propriété `HoldAllComplete` soit présente. * Aplatir les arguments de la `Sequence` s, sauf si `SequenceHold` est appliqué. * Appliquez les attributs `Flat` , `Listable` , `Orderless` selon le cas. * Appliquer l'évaluation associée aux valeurs de l'argument (leurs balises) * Appliquer l'évaluation associée à la tête.

Lorsque l'évaluation d'une expression est terminée, elle est marquée comme pleinement évaluée. Les évaluations ultérieures qui ont atteint cette expression ne tenteront pas de l'évaluer. Si une nouvelle règle est définie sur un symbole qui apparaît à l'intérieur, l'indicateur est supprimé et peut être réévalué. L'endroit notable que cet échec est avec `Condition` (`\;`): une règle conditionnelle pourrait ne pas s'appliquer à l'évaluation initiale. Si un symbole non lié change les valeurs et que la condition est maintenant applicable, l'expression est toujours marquée comme étant entièrement évaluée et ne changera pas en conséquence. La fonction `Update` est unique en ce sens qu'elle évalue son argument indépendamment de son état déjà évalué ou non, forçant un "vidage du cache".

Il existe un certain nombre d'autres fonctions souvent considérées comme exceptionnelles, telles que `Hold` , `Defer` , `ReplacePart` , `Extract` ou `ReleaseHold` . Ces effets peuvent tous être obtenus grâce à des attributs (tels que `HoldAll`) et à la définition de fonctions normales, et ils ne doivent pas nécessairement être traités de manière unique par l'évaluateur.

Examples

Application de `ReplaceAll` et` ReplaceRepeated`

Exemple de la manière dont `ReplaceAll` applique une règle au plus une fois, tandis que `ReplaceRepeated` le fait dans une boucle mais redémarre toujours l'application à partir de la première règle.

```
x + a /. {
  a_ + z :> (Print[0]; DoneA),
   a_ + x :> (Print[1]; y + z),
   a_ + y :> (Print[2]; DoneB)}

(* Prints "1", yields "y+z" *)

x + a //. {
  a_ + z :> (Print[0]; DoneA),
   a_ + x :> (Print[1]; y + z),
   a_ + y :> (Print[2]; DoneB)}

(* Prints "1", then prints "0", yields "DoneA" *)
```

Tri à bulles

Tri à bulles avec règles et remplacements:

```
list = {1, 4, 2, 3, 6, 7, 8, 0, 1, 2, 5, 4}

list //. {fsts___, x_, y_, lsts___} :> {fsts, y, x, lsts} /; y < x

(*
    Out[1] := {1, 4, 2, 3, 6, 7, 8, 0, 1, 2, 5, 4}
    Out[1] := {0, 1, 1, 2, 2, 3, 4, 4, 5, 6, 7, 8}
*)
```

Lire Ordre d'évaluation en ligne: https://riptutorial.com/fr/wolfram-mathematica/topic/6337/ordre-d-evaluation

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec wolfram-mathematica	Community, Karsten 7., Richard West
2	Ordre d'évaluation	Alex Meiburg, Kirill Belov