LEARNING wolfram-mathematica

Free unaffiliated eBook created from **Stack Overflow contributors.**

#wolfram-mathematic

a

Table of Contents

About

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

Chapter 1: Getting started with wolfram-mathematica

Remarks

This section provides an overview of what wolfram-mathematica is, and why a developer might want to use it.

It should also mention any large subjects within wolfram-mathematica, and link out to the related topics. Since the Documentation for wolfram-mathematica is new, you may need to create initial versions of those related topics.

Examples

What is (Wolfram) Mathematica?

Wolfram define Mathematica as "The world's definitive system for modern technical computing". A bold statement which is partially true. It's probably not the most predominant (as you have to pay quite a bit for commercial use) system so people use Python or R for example. What it is, is the most comprehensive **environment** for "technical computing" by providing the follow functionality:

- The Wolfram Language: A multi-paradigm language which covers symbolic computation with procedural, functional, list and rule-based programming
- "Notebooks": A combination of documentation, programs and results
- Wolfram Algorithmbase: Probably the largest curated set of algorithms covering most major areas of mathematics, computation, and graphics. Most people don't distinguish between Algorithmbase and the Language as they are so tightly intertwined
- With the introduction of Wolfram|Alpha came Wolfram Knowledgebase that covers many common areas of knowledge so you can answer a question such as "total time to defrost a 10 pound turkey in cold water" (which is 5h) in your program.

The whole system runs on the "Wolfram Engine" which is essentially a Virtual Machine, much like the Java Virtual Machine or Microsoft's Common Language Runtime allowing execution on a diverse set of platforms - currently Windows, Mac and Linux. Apart from running programs on a computer, you can also run them in the "Wolfram Cloud" which is a simple process compared to more "lower level" languages such as Java and C#.

The current version of Mathematica is 11 which can be run on the Desktop, Wolfram Cloud and iOS (both iPad and iPhone).

Read Getting started with wolfram-mathematica online: https://riptutorial.com/wolfram-mathematica/topic/3004/getting-started-with-wolfram-mathematica

Chapter 2: Evaluation Order

Remarks

This is supposed to explain the evaluation order as unambiguously as possible. It is probably less ideally suited as an introduction to Mathematica execution. It builds on Mathematica's `tutorial/Evaluation` page by explaining the order in which different rules are applied and explaining which functions are treated specially by the evaluation engine.

Evaluation Contexts

At any point, code is executing within a context. A context consists of a set of replacement rules (its dictionary), and a priority in which they should be evaluated. We group five kinds of contexts with distinct behavior and restrictions: *Set Context *Block Context *Matched Context *ReplaceAll Context *ReplaceRepeated Context

Set Contexts

A Set Context is that defined by some set of Set[] operations (the function more commonly written `=`). The main example is the primary execution environment. Its dictionary is modified whenever a Set[] is run on a variable that is not otherwise scopes. The other instances of Set Contexts arise from packages. A package context is related to its parent context in that any patterns within the package also become patterns within the parent context, with an appropriate prefix (a definition `foo[x_]=3*x` becomes `InnerPackageName\`foo[x_]=3*x`).

Set Contexts can only contain rules that have an associated "Tag", a string associated with the head to identify the rule's applicability more quickly. An application of `Set[yyy_,zzz_]` will determine a Tag by checking if `yyy` is a symbol. If it is, then it is the Tag. Otherwise, it checks `yyy[[0]]`, then `yyy[[0,0]]`, and so on. If it at some point this is determined to be a symbol, then that is taken as the Tag. If it is instead a non-symbol atom (such as String, Integer, Real...), then it will throw an error and no rule will be created.

The functions UpSet (written `^=`)and TagSet (written `/:=`) (and their cousins UpSetDelayed and TagSetDelayed) allow associating a rule with a different Tag. There is the still restriction that the Tag must be a symbol. UpSet will associate it with each of the arguments in the expression, or their head if they are a function with a symbol for a head. For instance, calling UpSet on `f[a,b,c+d,e[f,g],5,h[i][j][k],p_]` will associate the created rule with `a`, `b`, `e`, and `h`. The `c+d`, `5`, and `p_` arguments will have nothing associated with them, and will cause an error message to be displayed. The assignment will still succeed on each of the arguments, and (as will be made clear later in the evaluation order) it will still work for almost all purposes. TagSet is like UpSet, but you can specify exactly one symbol for the Tag. The symbol must still be something that could be set by Set or UpSet (a top-level symbol in the head or the arguments). For instance, `TagSet[f, f[a,b[c]], 2]` is acceptable and will associate the definition with `f`; `TagSet[a, f[a,b[c]], 2]` and `TagSet[b, f[a,b[c]], 2]` are also acceptable, but `TagSet[c, f[a,b[c]], 2]` is not.

Rules inside a Context need an application priority, since there can be many rules that apply to a given expression. (This is also true in ReplaceAll and ReplaceRepeated Contexts, but they handle it very differently). The priority is generally intended to correspond to the *specificity* of the pattern. Given an expression `a[q][b[c,d],e[f,g]]` to evaluate, with the head and arguments all evaluated as fully as they will be (see below TODO), begin by looking for rules tagged with `b` that apply; then rules tagged with `e`; then rules tagged with `a`. Within each set of rules, there is maintained an order on those associated with a given symbol. Rules with no blanks (such as `f[a,b]=3`) are automatically placed at the top and sorted in canonical order (the order of Sort). Each time a new rule is added, the kernel will go through the list; if some rule has the exact same LHS, then it gets replaced in-place. Otherwise, it does a specificity comparison. If a rule X already in the list is determined to be "less specific" than the new rule Y, then Y is placed immediately before X. Otherwise, it continues through the list. If no rule is less specific, then the rule is placed at the end of the list. The specificity checkinng is more complicated and given in more detail in the section below.

Rule Specificity

*If two expressions have no instance of BlankSequence (__), BlankNullSequence (___), Optional (:), Alternatives (|), Repeated (..), RepeatedNull (...), or optional arguments (_.), then they can be compared structurally. Given two equivalent expression trees X and Y, where all blanks in Y are also blanks in X, but X has blanks where Y does not, then Y is more specific. *If two expressions are equivalent except that some instances of _ have been replaced with __ in the other expression, or __ have been replaced withh ___, then the former is more specific. *If stripping one more Optional (:) or optional (_.) terms gives the other expression, then the latter is more specific. *If a certain set of choices from Alternatives gives the other expression, then the latter is more specific. *If replacing all instances of `RepeatedNull[foo]` with `Repeated[foo]`, or `Repeated[foo]` with `foo`, gives the other expression then the latter is more specific *Some combations of these rules can be applied at once, but it's not currently know what the cases for this are. *Combinations of expression such as `_List` and `{___}` theoretically should treat them identically, but the comparison appears to be strangely context-dependent, occasionally ranking them one way or the other.

Block Context

A Block Context is more restrictive, in that the LHS of a rule in a Block can only be a symbol. That is, only definitions of the form `f=2+x`, not `f[x_]=2+x`. (Note that, from a practical standpoint, functions can still be constructed with definitions such as `Set[Block is related to its parent context in that any new definitions during the evaluation of the Block get forwarded to the surrounded context as normal, but it will "shadow" some set of variables, providing definitions that can hide those of the surrounding context. Definitions from the surrounding context are still accessible during evaluation of the inner expression. Because there can only be definition associated with a symbol, there is no notion of priority as above.

Matched Context

After a rule has been matched, there are locally bound definitions for variables. This occurs

lexically. That is to say, it subsitutes in the bound definitions for variables in the expression, without evaluating anything else. Only once all subsitutions have occurred does it begin evaluating the expression, as a whole, from the top again. The primary way Matched Contexts are created is a rule from a Set Context or Rule. For instance, in

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

Upon matching the `f[x_]` rule to `f[y]`, the symbol `x` is bound to the value `x^2`. It performs the one substitution, but because it does not evaluate `g`, it returns `x^2+g[1]`. This is then evaluated in the surrounding Set Context again and becomes `1+x+x^2`. The significant difference in evaluation in a Matched Context is that the replacement is *not* recursive. When it subsitutes `x->x^2`, it does not repeat even on its own results.

Matched Contexts are also created by `With`, `Module`, `Function`, and `Replace` notably. Many other functions create them internally, for instance `Plot` uses this type of context in evaluating the expression to be plotted.

ReplaceRepeated Context

A ReplaceRepeated Context is created when any application of `ReplaceRepeated` occurs. This is distinct in that it can have any expression as a rule LHS, including those with no tag, such as `_[_]`. In this sense it is the most flexible context. It can also include several rules which can conflict, so it must maintain a priority. A ReplaceRepeated Context will apply the first rule in the list first wherever applicable. If it fails to match, it proceeds to the second, and so on. If at any point a rule matches, it returns to the first rule and begins again. If at any point a rule application occurs and no change occurs, it will exit -- even if other rules later in the list would make a change. This means that any less specific rules earlier in the list will prevent later rules from ever being used. Additionally, placing `a_->a` at the front of the rule list will cause an immediate termination of the entire `ReplaceRepeated`.

ReplaceAll Context

A ReplaceAll Context is created when any application of `ReplaceAll` occurs. Its functioning is similar to that of ReplaceRepeated in that its rule application priority goes in order in the list when two can both apply at the same level of the expression. However, it is like a Matched Context in that the replaced contents are not evaluated further, *not even by later rules*. For instance `x/.{x->y,y->z}` yields `y`. Thus it is incorrect to view an application of ReplaceAll has applying each rule in turn. Instead, it traverses the tree, looking for applicable rules. When it finds something that matches, it executes the replacement, then returns up the tree, without traversing the new tree. It is also worth noting that it attempts to apply rules from the top down, possibly going out of order of the list as a result. For instance,

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

both yield `5` without printing anything. Because the `Cos[_]` matches a higher level of the tree, it applies that one first.

`Hold` and `Evaluate` and the execution order

The order of evaluation, given an expression, proceeds as: *Evaluate the head as thoroughly as possible *If the head has a `Hold` property (`HoldFirst`, `HoldRest`, `HoldAll`, `HoldAllComplete`), then *Check the relevant arguments. Unless it's `HoldAllComplete`, check if the head is `Evaluate`. If it is, then strip the `Evaluate` and mark it to be evaluated anyway. *Check arguments for instances of `Unevaluated` and strip them as necessary, unless the property `HoldAllComplete` is present. *Flatten arguments from `Sequence`s, unless `SequenceHold` is applied. *Apply the attributes `Flat`, `Listable`, `Orderless` as applicable. *Apply evaluation associated with the argument's upvalues (their Tags) *Apply evaluation associated with the head.

When evaluation of an expression is complete, it is marked as fully evaluated. Subsequent evaluations that hit that expression will not try to evaluate it. If a new rule is defined on a symbol that occurs inside, then the flag is removed, and it can be evaluated again. The notable place that this 'fails' is with `Condition` (`\;`): a conditional rule could possibly not apply upon the initial evaluation. If an unrelated symbol changes values and now the condition is applicable, the expression is still marked fully evaluated, and will not change as a result. The function `Update` is unique in that it will evaluate its argument regardless of its already-evaluated status or not, forcing a "cache flush" of sorts.

There are a number of other functions that are often regarded as exceptional, such `Hold`, `Defer`, `ReplacePart`, `Extract`, or `ReleaseHold`. These effects can all be achieved through attributes (such as `HoldAll`) and normal function definition, and do not need to be handled uniquely by the evaluator.

Examples

Application of `ReplaceAll` and `ReplaceRepeated`

Example of how `ReplaceAll` only applies a rule at most once, while `ReplaceRepeated` will do so in a loop but always restart application from the first rule.

```
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" *)
```

Bubble sort

Bubble sorting with rules and replacements:

```
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}
*)
```

Read Evaluation Order online: https://riptutorial.com/wolfram-mathematica/topic/6337/evaluation-order

Credits

S. No	Chapters	Contributors
1	Getting started with wolfram-mathematica	Community, Karsten 7., Richard West
2	Evaluation Order	Alex Meiburg, Kirill Belov