The Mathematica® Journal

Demystifying Rules

Nancy Blachman

President
Variable Symbols, Inc.
356 Bush Street
Mountain View, CA 94041
Fax: 650-966-8998
Tel: 650-966-8999
Nancy@VariableSymbols.com
www.VariableSymbols.com

Have you ever wondered in what order Mathematica applies replacement rules or why Mathematica sometimes applies a replacement rule and other times doesn't? This article explains how replacement rules work and why they are useful.

■ Replacement Rules

Rules are a mechanism for replacing a variable with a value in an expression. The **`ReplaceAll`** operator, which is abbreviated **`/.`** (with no space between the **`/`** and the **`.`**), applies a rule or list of rules after evaluating the expression. Think of **`/.`** as meaning *given that* and **`->`** (a minus sign together with a greater than sign, with no space in between the two) as meaning *goes to*. So the expression **`2x+3y/.x->5b`** means **`2x+3y`** *given that* **`x`** *goes to* **`5b`**.

```
2 x + 3 y /. x -> 5 b

10 b + 3 y
```

© 2002 Wolfram Media, Inc.

Notice that all **x**'s in the expression to the left of **/.** are replaced by **5b**, but that **x** has not been assigned a value.

```
x

x
```

Here are some more examples of replacement rules, each followed by an explanation of the replacement.

```
2 x + 3 y /. {x -> 7, 3 y -> 5 a}

14 + 5 a
```

The symbol **x** gets replaced by **7** and the expression **3y** gets replaced by **5a**, so the expression **2x+3y** is transformed into **2*7+5a**, which evaluates to **14+5a**.

```
2 x + 3 y /. {x -> a, 2 x -> b}

b + 3 y
```

Mathematica uses a greedy algorithm. It looks for the largest expression that matches a pattern. First *Mathematica* checks whether there is a rule that matches the entire expression. If not, it checks whether there is a rule that matches part of an expression. In this case, the rule **2x->b** matches part of the expression **2x+3y**, so the rule is applied. Another way to look at this is that *Mathematica* matches the most specific expression, and the rule **2x->b** is more specific than the rule **x->a**.

```
2 x + 3 y /. {x -> a, x -> b}

2 a + 3 y
```

Rules of equal specificity are applied from left to right, so the rule **x->a** is applied before **x->b**.

```
2 x + 3 y /. {x -> b, x -> a}
```

```
2 b + 3 y
```

If we put the rule **x->b** first, it is applied before **x->a**.

```
2 x + 3 y /. {x -> b, 2 x -> a}
```

```
a + 3 y
```

The rule **2x->a** is applied here because it matches a larger piece of the entire expression than the rule **x->b**.

```
2 x + 3 y /. {{x -> a}, {x -> b}}
```

```
{2 a + 3 y, 2 b + 3 y}
```

When *Mathematica* is given a list of lists of rules, each sublist is applied to the original expression. The result is a list of transformed expressions.

```
1 / Sqrt[x^2] /. Sqrt[x^2] -> 9
```

$$\frac{1}{\sqrt{x^2}}$$

Since *Mathematica* doesn't know anything about **x^2** (whether it is positive or negative), it leaves the expression **Sqrt[x^2]** unchanged. *Mathematica* internally represents **1/Sqrt[x^2]** as **(x^2)^-(1/2)** and **Sqrt[x^2]** as **(x^2)^(1/2)**.

```
FullForm[1 / Sqrt[x^2]]
```

```
Power[Power[x, 2], Rational[-1, 2]]
```

```
FullForm[Sqrt[x^2]]

Power[Power[x, 2], Rational[1, 2]]
```

Consequently, *Mathematica* can't find the expression **Sqrt[x^2]** in the original expression **1/Sqrt[x^2]**. Therefore, the rule isn't applied and *Mathematica* returns the original expression unchanged.

```
Sqrt[x] + 1 / Sqrt[x] /. {Sqrt[x] -> a, x -> b}

a + 1/√b
```

Because **Sqrt[x]** doesn't appear in the expression **1/Sqrt[x]**, the first rule (**Sqrt[x]->a**) is only applied to the first element of the sum (**Sqrt[x]**). The second rule (**x->b**) is applied to the second element (**1/Sqrt[x]**).

```
{1, 11, 111, 1111} /. {1 -> one}

{one, 11, 111, 1111}
```

Numbers are atomic expressions and can't be subdivided. The rule **1 -> one** is applied only when **1** matches the entire number.

```
2 x + 3 x /. {2 x -> y, 3 x -> 3}

5 x
```

Mathematica evaluates the expression to the left of the **/.** before applying any of the rules. The expression **2x+3x** evaluates to **5x** and then none of the rules match, so they aren't applied.

```
2 x + 3 y /. {2 x -> a, a -> b}

a + 3 y
```

Mathematica applies only those rules that apply to the original expression, so **2x** gets replaced by **a**. No other rule applies to the original expression.

```
2 x + 3 y //. {2 x -> a, a -> b}

b + 3 y
```

The notation **//.** is an alias for **ReplaceRepeated**. The rules are applied repeatedly until there are no more rules that apply. On the first pass, **2x** gets replaced by **a**. On the second pass, **a** gets replaced by **b**.

```
{a, b, c, d} /. {a -> b, b -> c, c -> b, d -> a}

{b, c, b, a}
```

The symbol **a** gets replaced by **b**, the symbol **b** gets replaced by **c**, and the symbol **c** gets replaced by **a**.

```
{a, b, c, d} //. {a -> b, b -> c, c -> d, d -> a}

ReplaceRepeated::rrlim :
  Exiting after {a, b, c, d} scanned 65536 times.

{a, b, c, d}
```

The rules are applied repeatedly. After applying the result many, many times, *Mathematica* stops so that it won't get caught in an infinite loop.

■ Using Rules

Let's look at some examples of functions that return results in terms of rules. **Solve** finds the solutions to a polynomial equation of degree four or less (as well as some polynomial equations of higher degree). The solution is stated as a list of replacement rules.

```
soln = Solve[x^3 - 3 x^2 - 17 x + 51 == 0, x]
```

$$\{\{x \to 3\}, \{x \to -\sqrt{17}\}, \{x \to \sqrt{17}\}\}$$

We can get a list of the values for **x** by using **ReplaceAll**.

```
x /. soln
```

$$\{3, -\sqrt{17}, \sqrt{17}\}$$

FindRoot, **DSolve**, and **NSolve** also return solutions in terms of replacement rules.

```
FindRoot[Sin[x] / x, {x, 3, 4}]
```

$$\{x \to 3.14159\}$$

```
DSolve[y''[x] + 2 y'[x] == 0, y[x], x]
```

$$\left\{\left\{y[x] \to -\frac{1}{2} e^{-2x} C[1] + C[2]\right\}\right\}$$

```
p = x^10 + x^9 - x^7 - x^6 - x^5 - x^4 - x^3 + x + 1;
```

```
complexRoots = NSolve[p == 0]

{{x → -0.943305 - 0.331928 i},
 {x → -0.943305 + 0.331928 i},
 {x → -0.734438 - 0.678676 i},
 {x → -0.734438 + 0.678676 i},
 {x → -0.292332 - 0.956317 i},
 {x → -0.292332 + 0.956317 i},
 {x → 0.456866 - 0.889536 i}, {x → 0.456866 + 0.889536 i},
 {x → 0.850137}, {x → 1.17628}}
```

We can easily compute the absolute values of the roots.

```
Abs[x] /. complexRoots

{1., 1., 1., 1., 1., 1., 1., 1., 0.850137, 1.17628}
```

What do the absolute values of the complex roots have in common?

The function **DispersionReport**, from the standard package **Statistics`DescriptiveStatistics`**, returns a list of statistics in the form of rules.

```
Needs["Statistics`DescriptiveStatistics`"]
```

```
dReport = DispersionReport[{3, 7, 2, 3, 1, 2, 8}]

{Variance → 152/21, StandardDeviation → 2 Sqrt[38/21],
 SampleRange → 7, MeanDeviation → 106/49,
 MedianDeviation → 1, QuartileDeviation → 2}
```

We can pick out a particular result or several results from the list.

```
MeanDeviation /. dReport
```

$$\frac{106}{49}$$

```
{Variance, StandardDeviation} /. dReport
```

$$\left\{\frac{152}{21}, 2\sqrt{\frac{38}{21}}\right\}$$

Graphics options are specified in terms of rules.

```
Options[Plot]
```

```
{AspectRatio → 1/GoldenRatio, Axes → Automatic,
 AxesLabel → None, AxesOrigin → Automatic,
 AxesStyle → Automatic, Background → Automatic,
 ColorOutput → Automatic, Compiled → True,
 DefaultColor → Automatic, Epilog → {}, Frame → False,
 FrameLabel → None, FrameStyle → Automatic,
 FrameTicks → Automatic, GridLines → None,
 ImageSize → Automatic, MaxBend → 10.,
 PlotDivision → 30., PlotLabel → None, PlotPoints → 25,
 PlotRange → Automatic, PlotRegion → Automatic,
 PlotStyle → Automatic, Prolog → {}, RotateLabel → True,
 Ticks → Automatic, DefaultFont :→ $DefaultFont,
 DisplayFunction :→ $DisplayFunction,
 FormatType :→ $FormatType, TextStyle :→ $TextStyle}
```

```
PlotPoints /. Options[Plot]
```

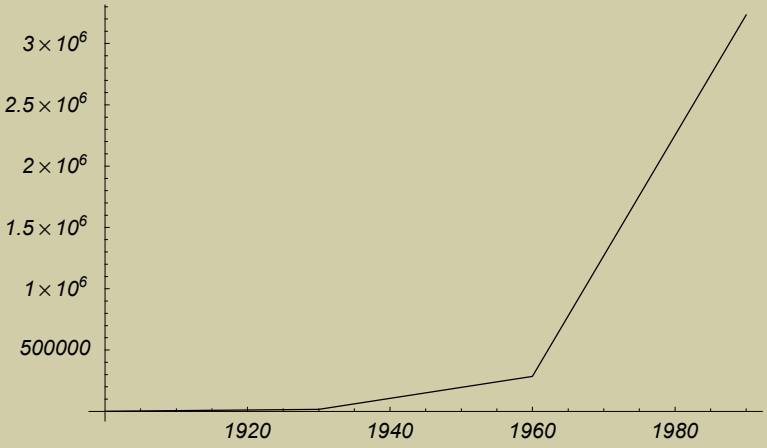
```
25
```

■ Naming Parts of Expressions

Rules provide a mechanism for assigning names to parts of an expressions and then specifying what to do with some or all the names.

```
nationalDebt = {{1900, 1263},
   {1930, 16185}, {1960, 284093}, {1990, 3233313}};
```

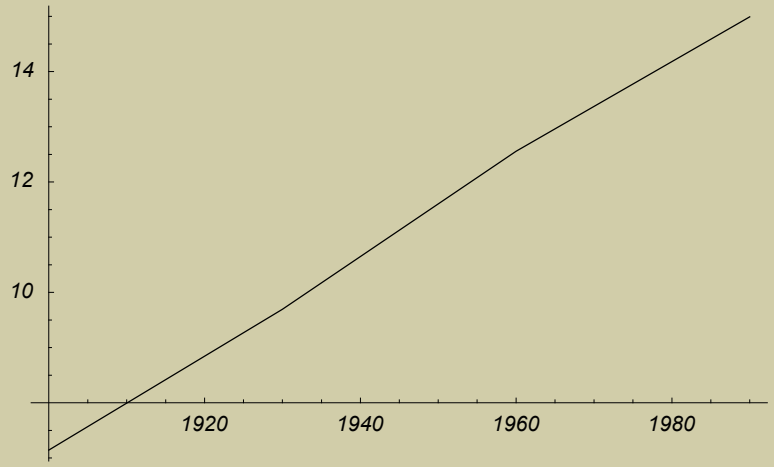
```
ListPlot[nationalDebt, PlotJoined -> True];
```



The notation **year_** is a named pattern that designates a *Mathematica* expression. The rule {**year_,debt_}->{year,Log[debt]}** specifies that given a pair of values, *Mathematica* assigns the name **year** to the first element and **debt** to the second element in the pair, and returns the pair **{year,Log[debt]}**.

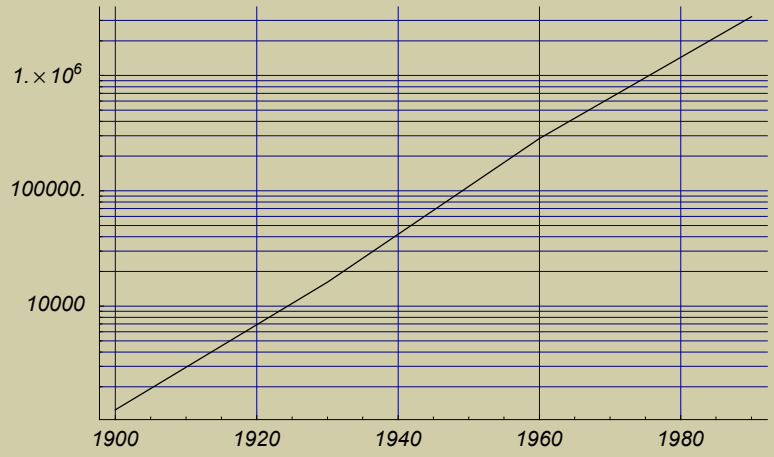
```
nationalDebt /. {year_, debt_} -> {year, Log[debt]}

{{1900, Log[1263]}, {1930, Log[16185]},
 {1960, Log[284093]}, {1990, Log[3233313]}}
```

```
ListPlot[%, PlotJoined -> True];
```



Actually, we didn't need to use this rule because there's a function for making log plots.

```
Needs["Graphics`Graphics`"];
LogListPlot[nationalDebt,
  PlotJoined -> True, GridLines -> Automatic];
```



Let's look at some more examples of rules with named patterns.

```
{{1, 2, 3}, {4, 5, 6}} /. {x_, y_, z_} -> {x, Log[y]}

{{1, Log[2]}, {4, Log[5]}}
```

Triples are replaced by pairs.

```
{{1, 2}, {3, 4}, {5, 6}} /. {x_, y_} -> {y, x}

{{2, 1}, {4, 3}, {6, 5}}
```

The rule swaps the elements in each pair.

```
{{1, 2, 3}, {4, 5, 6}} /. {x_, y_} -> {y, x}

{{4, 5, 6}, {1, 2, 3}}
```

The elements in a pair are swapped. There is only one pair in the expression.

```
{{1, 2}, {3, 4}, {5, 6}} /. {x_, y_} -> {x, Log[y]}

{{1, Log[2]}, {3, Log[4]}, {5, Log[6]}}
```

Each pair gets replaced by another pair consisting of the first element and the log of the second element.

```
{{1, 2}, {3, 4}} /. {x_, y_} -> {x, Log[y]}

{{1, 2}, {Log[3], Log[4]}}
```

The largest (outermost) pair is replaced by another pair consisting of the first element and the log of the second element.

```
f[a + f[b]] /. f -> g

g[a + g[b]]
```

The symbol **f** is replaced by **g**.

```
f[a + f[b]] /. f[x_] -> g[x]

g[a + f[b]]
```

Since the rule matches the entire expression, *Mathematica* replaces the head of the expression with **g**. It doesn't look deeper into the expression.

```
f[a + f[b]] //. f[x_] -> g[x]

g[a + g[b]]
```

Mathematica repeatedly replaces **f** with **g** until the rule no longer applies.

```
Log[x] + Log[y + Log[z + Log[t]]] /. Log[w_] -> w

x + y + Log[z + Log[t]]
```

Mathematica applies the rule to the subexpressions **Log[x]** and **Log[y + Log[t]]** and transforms them to **x** and **y+Log[z+Log[t]]**. The rule doesn't get applied to subexpressions of these expressions. Once a transformation is made, *Mathematica* doesn't apply the rule again, unless we use **//.** or **ReplaceRepeated**. So, the rule doesn't get applied to the nested calls to **Log**.

```
Log[x] + Log[y + Log[z + Log[t]]] //. Log[w_] -> w

t + x + y + z
```

■ Delayed Rules

The notation **->** is the alias for the function **Rule**. The notation **:>** is the alias for the function **RuleDelayed**. The notation **x->Random[]** specifies a rule to replace **x** with the evaluated **Random[]**. The expression **Random[]** evaluates to a random number. The notation **x:>Random[]** specifies a rule to replace **x** with the unevaluated expression **Random[]**.

```
Table[x, {3}]

{x, x, x}
```

```
Table[x, {3}] /. x -> Random[]

{0.44115, 0.44115, 0.44115}
```

```
Table[x, {3}] /. x :> Random[]

{0.384776, 0.609794, 0.790723}
```

Delayed rules are like delayed assignments in that the right side of the rule is evaluated after the replacement has been made.

Using **Trace**, you can see the order in which *Mathematica* evaluates these expressions. With an immediate assignment, *Mathematica* first replaces **Table** with a list of three **x**'s. Then *Mathematica* calls **Random** and obtains a value. Then each **x** in the list is replaced by that value.

© 2002 Wolfram Media, Inc.

```
Table[x, {3}] /. x -> Random[] // Trace // ColumnForm

{Table[x, {3}], {x, x, x}}
{{Random[], 0.39422}, x → 0.39422, x → 0.39422}
{x, x, x} /. x → 0.39422
{0.39422, 0.39422, 0.39422}
```

With a delayed assignment, *Mathematica* first replaces **Table** with a list of three **x**'s. Then *Mathematica* replaces each **x** in the list with a call to **Random**. Then *Mathematica* evaluates each call to **Random**.

```
Table[x, {3}] /. x :> Random[] // Trace // ColumnForm

{Table[x, {3}], {x, x, x}}
{x :→ Random[], x :→ Random[]}
{x, x, x} /. x :→ Random[]
{Random[], Random[], Random[]}
{Random[], 0.67701}
{Random[], 0.21584}
{Random[], 0.0830972}
{0.67701, 0.21584, 0.0830972}
```

You can specify rules to transform sums of angles. These rules used to be in the standard package **Algebra`Trigonometry`**. In Version 3, they were incorporated into the *Mathematica* kernel.

```
sumsOfAngleRules = {
 Sin[x_ + y_] :> Sin[x] Cos[y] + Sin[y] Cos[x],
 Cos[x_ + y_] :> Cos[x] Cos[y] - Sin[x] Sin[y],
 Tan[x_ + y_] :>
   (Tan[x] + Tan[y]) / (1 - Tan[x] Tan[y]) }

{Sin[x_ + y_] :→ Sin[x] Cos[y] + Sin[y] Cos[x],
 Cos[x_ + y_] :→ Cos[x] Cos[y] - Sin[x] Sin[y],
 Tan[x_ + y_] :→ (Tan[x] + Tan[y]) / (1 - Tan[x] Tan[y])}
```

■ Exercises

I hope you now understand how rules work and how you can use them to manipulate your expressions. Here are some exercises to test your understanding.

1. For the expression

```
a = (x + y)^2 Sin[2 (x + 3 y)];
```

write a rule that expands the argument of **Sin**, but does not expand the term **(x+y)^2**.

Solution:

```
a /. Sin[x_] :> Sin[Expand[x]]
```

$$(x + y)^2 \sin[2 x + 6 y]$$

2. In the substitution

```
y (x + 2) (x - 2) /. y z_ -> y Expand[z]
```

$$(-2 + x)\,(2 + x)\,y$$

why doesn't the rule expand the term **(x+2)(x-2)**? Write a rule that will expand **y(x+2)(x-2)**.

Solution: The right-hand side of the rule gets evaluated immediately, so **Expand[z]** evaluates to **z**. Use a delayed rule instead.

```
y (x + 2) (x - 2) /. y z_ :> y Expand[z]
```

$$(-4 + x^2)\,y$$

3. Given five points $\{\{x_1, y_1\}, \ldots, \{x_5, y_5\}\}$ and five heights $\{z_1, \ldots, z_5\}$ at those points, construct triplets of the form $\{\{x_1, y_1, z_1\}, \ldots, \{x_5, y_5, z_5\}\}$ using only list manipulation constructs and rules, and without using **Part** or functions with iterators (such as **Table** or **Do**).

Solution:

```
xyPairs = Transpose[{Array[x, {5}], Array[y, {5}]}]

{{x[1], y[1]}, {x[2], y[2]},
 {x[3], y[3]}, {x[4], y[4]}, {x[5], y[5]}}
```

```
zValues = Array[z, {5}]

{z[1], z[2], z[3], z[4], z[5]}
```

```
Transpose[{xyPairs, zValues}] /.
 {{x_, y_}, z_} -> {x, y, z}

{{x[1], y[1], z[1]},
 {x[2], y[2], z[2]}, {x[3], y[3], z[3]},
 {x[4], y[4], z[4]}, {x[5], y[5], z[5]}}
```

4. **FactorInteger** returns a list of the prime factors of its argument together with their exponents. For example,

```
FactorInteger[909]

{{3, 2}, {101, 1}}
```

If you raise the first number in the pair to the power of the second number and then multiply the results together, you should obtain the number you factored.

```
3^2 101^1

909
```

Write a rule that takes the result returned by **FactorInteger** and computes the original number.

Solution:

```
factors = FactorInteger[90909]

{{3, 3}, {7, 1}, {13, 1}, {37, 1}}
```

```
factors /. {f_Integer, e_} -> f^e

{27, 7, 13, 37}
```

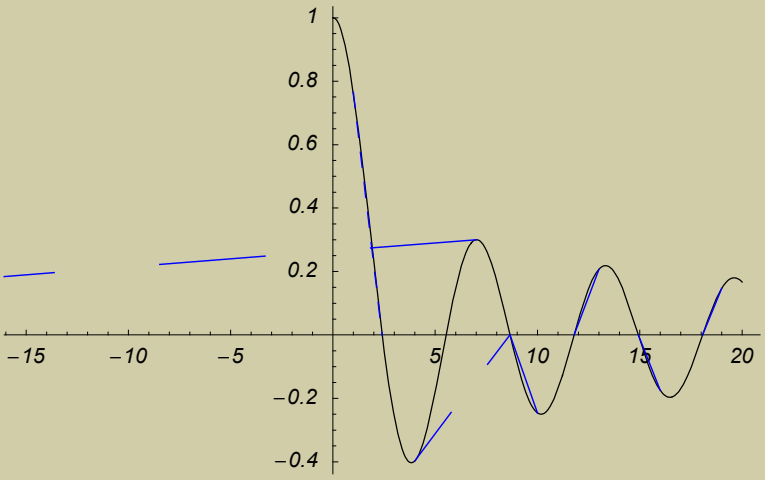
```
Apply[Times, %]

90909
```

```
FactorInteger[9090909] /. {f_Integer, e_} -> f^e //
 Apply[Times, #] &

9090909
```

5. **FindRoot** uses Newton's method to obtain a root, given a function and a starting value. Plot the function **BesselJ[0,x]** for x in the range [0, 20]. Choose seven starting values of x in that range and, for each one, draw the line segment that joins the point **{x,BesselJ[0,x]}** on the graph to the point **{r,0}** on the x-axis, where **r** is the root returned by **FindRoot** when given the starting value **x**. (You can combine a graph of a function with a line produced with the command **Graphics** by using **Show**.)

Solution:

```
Show[Plot[BesselJ[0, x],
   {x, 0, 20}, DisplayFunction -> Identity],
    Graphics[{Thickness[0.05], RGBColor[0, 0, 1],
            Table[{Thickness[0.002], Dashing[
       {x0 / 50.}], Line[{{x0, N[BesselJ[0, x0]]},
        { x /. FindRoot[BesselJ[0, x], {x, x0}],
         0}}]}, {x0, 1, 19, 3}]
        }],
    DisplayFunction :> $DisplayFunction];
```



About the Author

Nancy Blachman is president and founder of Variable Symbols, Inc., a company that specializes in training and consulting in *Mathematica* and Linux. She is author of the popular tutorial book *Mathematica: A Practical Approach* (Prentice Hall). From 1987–1989, she taught "Problem Solving with *Mathematica*" at Stanford University.