

Some Rule Based Programming Examples

Three examples are given of *Mathematica* programs that use replacement rules rather than procedural constructs. The diversity of the applications demonstrates the power and flexibility of this approach to programming.

J. M. Selig

In the first edition of Stephen Wolfram's *Mathematica* book is an example of a rule-based program that implements the bubble sort algorithm. In this work, we present three more examples of rule-based programs: an implementation of a Turing machine, a function to solve tri-diagonal systems of linear equations by Gaussian elimination, and an implementation of a finite group from a presentation. The idea is to demonstrate the power and flexibility of this style of programming.

We begin by discussing some important generalities concerning rules in *Mathematica*. First, what is the difference between `Rule` (`->`) and `RuleDelayed` (`:>`)? The statement `lhs -> rhs` causes *Mathematica* to evaluate `rhs` immediately, while `lhs :> rhs` evaluates `rhs` only when the rule is applied. Hence, for simple re-arrangements we can use `->`. If the `rhs` contains expressions that depend on values in the `lhs` and that may change from case to case, we must use `:>`.

Although we often talk about a set of rules, *Mathematica* stores rules in lists. When using `Replace` or `ReplaceAll` (`/.`), the order that the rules appear in the list is important, since each rule is tried in turn beginning with the first. Sometimes we can use this fact to our advantage by placing a more general rule after a more specific one. For example, consider the function defined by applying the following pair of rules:

```
In[1]:= delta[i_, j_]:= {i, j} /. {{x_, x_} -> 1, {__} -> 0}
```

The first rule returns 1 if the two arguments are the same, the second rule returns 0 whatever the argument. Since the first rule is applied first, the second rule is only applied if the first fails to match. The result is that the function returns 1 if the two arguments are the same and 0 otherwise.

```
In[2]:= delta[3, 3]
```

```
Out[2]= 1
```

```
In[3]:= delta[3, 4]
```

```
Out[3]= 0
```

Notice that `ReplaceAll` (`/.`) will apply at most one of the rules on the list, so if one rule finds a match, the rules after it are ignored.

J. M. Selig is senior lecturer at SouthBank University, London.

Next, we come to the order of evaluation for the function `ReplaceRepeated` (`//.`). This function is equivalent to repeatedly applying `ReplaceAll` (`/.`) to the arguments. The first rule in the list is repeatedly applied until it no longer matches, then the second rule is applied, and so forth. (It is not true that each rule of the list is applied in turn.) We can illustrate this property with the following example:

```
In[4]:= {a, b, c} //. {{a, any___, _} -> {a, any},
                  {___, b, ___} -> no, {a} -> yes}
```

```
Out[4]= yes
```

The first rule removes the last element of any list beginning with `a`. The second rule returns `no` if the list contains `b`. The last rule returns `yes` if the list contains the single element `a`. What happens when these rules are applied repeatedly to the list `{a, b, c}`? The first rule matches since the list starts with `a` and so it returns the list `{a, b}`. The rules are applied again, and again the first rule find a match, this time returning `{a}`. The next time the rules are applied, neither the first nor the second rule find a match and so the third rule is applied and returns `yes`. The final application of the list of rules finds no matches and hence the evaluation terminates, returning `yes`. The point here is that the second rule never gets applied successfully since the `b` in the list is removed before the second rule "gets a chance to look at it."

Finally, we mention conditional statements. The application of a rule can be controlled by the function `Condition` (`/;`).

```
In[5]:= rule = {a_, b_} :> {b, a} /; b < a ;
```

```
In[6]:= {1, 2} /. rule
```

```
Out[6]= {1, 2}
```

```
In[7]:= {4, 3} /. rule
```

```
Out[7]= {3, 4}
```

Applying this type of rule repeatedly gives a one-line version of the bubble sort algorithm:

```
In[8]:= sort[{{some__}}]:= {some} //.
          {{any1___, a_, b_, any2___} :> {any1, b, a, any2} /; b<a}
```

```
In[9]:= sort[{9,2,7,4,6,7}]
```

```
Out[9]= {2, 4, 6, 7, 7, 9}
```


To display all the stages in the process, we use the following function :

```
In[3]:= showTape[tape_]:=
  (temp = tape /. rules;
   Print[temp];
   If[Head[temp] === List, showTape[temp], temp])
```

We can simplify the input expression by using the function `Characters`:

```
In[4]:= bracketTest[string_String]:=
  showTape[Join[{s}, Characters[string]] ]
```

For example:

```
In[5]:= bracketTest["()()"]
  {(, s1, ), (, (, ), )}
  {(, s2, (, (, (, ), )}
  {s2, (, (, (, ), )}
  {s, (, (, ), )}
  {(, s1, (, ), )}
  {(, (, s1, ), )}
  {(, (, s2, (, )}
  {(, s2, (, (, )}
  {s2, (, (, (, )}
  {s, (, )}
  {(, s1, )}
  {(, s2, ()}
  {s2, (, ()}
  {s}
  success
```

```
In[6]:= bracketTest["()()()"]
  {(, s1, ), ), (, (, ), )}
  {(, s2, (, ), (, (, ), )}
  {s2, (, (, ), (, (, ), )}
  {s, ), (, (, ), )}
  fail
```

Of course, this is not the most efficient way to detect valid bracket expressions. *Mathematica*'s lists are much more flexible than a Turing tape, so there is a much simpler set of rules that will do the job. We can remove matching pairs of open and closed brackets in a single step and we do not need the starting state at the beginning of the list:

```
In[7]:= rules2 = {
  {any1___, "(", ")", any2___} -> {any1, any2},
  {} -> success,
  {__} -> fail };
```

```
In[8]:= Characters["()()"] //. rules2
```

```
Out[8]:= success
```

The point of this example is that any Turing machine can be implemented by applying a set of rules to a list in *Mathematica*. So, anything that can be done by a procedural program can also be done using replacement rules. Given such a choice, it seems sensible to use the style that gives the simpler, more easily understood program for each particular problem. The surprise is that this is often the rule-based approach.

Tridiagonal Systems of Linear Equations

In the next example, we use replacement rules to solve tridiagonal systems of linear equations, that is, equations given by $Mx = r$, where M is a tridiagonal matrix. Let a , b , and c be lists of coefficients: b is the list of diagonal entries of M , a is the list of sub-diagonal entries, and c contains the super-diagonal entries. The list r contains the right-hand-sides of the equations. Our program performs Gaussian elimination by applying rules to $\{a, b, c, r\}$. As usual with Gaussian elimination, there are two parts to the program: conversion to upper-triangular form, followed by the solution by back substitution.

The first rule initializes a list that holds all the data and the results of the calculation so far:

```
In[1]:= rule1 =
  {a_, {b1_, brest___}, {c1_, crest___}, {r1_, rrest___}} ->
  {a, {brest}, {crest}, {rrest}, {b1, c1, r1}};
```

This rule is applied only once. The last element of the resulting list contains the nonzero part of the first row of the upper-triangular matrix.

The next rule repeatedly appends the next row of the upper-triangular matrix and removes the "used" coefficients:

```
In[2]:= rule2 =
  {{aj_, arest___}, {bj_, brest___}, {cj_, crest___},
   {rj_, rrest___}, x___, {bi_, ci_, ri_}} ->
  {{arest}, {brest}, {crest}, {rrest}, x, {bi, ci, ri},
   {bj - aj*ci/bi, cj, rj - aj*ri/bi}};
```

When no more c coefficients remain, this rule no longer matches and the next rule is applied:

```
In[3]:= rule3 =
  {{aj_}, {bj_}, {}, {rj_}, x___, {bi_, ci_, ri_}} ->
  {(bi*rj - ri*aj)/(bi*bj - ci*aj), x, {bi, ci, ri}};
```

This rule can also be thought of as an initialization and only applies once. The last equation is solved and the result is placed at the beginning of the list. Now the beginning of the list contains the solutions so far and the end contains the unused rows of the upper-triangular matrix.

The last rule repeatedly prepends the next solution to the list and removes the last row of the upper-triangular matrix:

```
In[4]:= rule4 =
  {yj_, x___, {bi_, ci_, ri_}} ->
  {(ri - ci*yj)/bi, yj, x};
```

Finally, when all the rows of the upper-triangular matrix have been used, the list contains solutions for all the unknowns and nothing else, so the rule no longer matches.

Our function applies these rules after checking that the input lists have the right lengths:

```
In[5]:= TriSolve[a_, b_, c_, r_] :=
  ({a,b,c,r} //. {rule1, rule2, rule3, rule4}) /.
  Length[a]+1 == Length[b] == Length[c]+1 == Length[r]
```

Here are a few examples:

```
In[6]:= TriSolve[{1,2,3,4}, {1,2,2,2,1}, {1,2,3,4}, {1,1,1,1,1}]
Out[6]= { $\frac{9}{19}, \frac{10}{19}, -\frac{5}{19}, \frac{3}{19}, \frac{7}{19}$ }

In[7]:= TriSolve[ Table[-1, {9}], Table[2, {10}],
Table[-1, {9}], {1,0,0,0,0,0,0,0,0}]
Out[7]= { $\frac{10}{11}, \frac{9}{11}, \frac{8}{11}, \frac{7}{11}, \frac{6}{11}, \frac{5}{11}, \frac{4}{11}, \frac{3}{11}, \frac{2}{11}, \frac{1}{11}$ }

In[8]:= TriSolve[{1,1}, {2,2,2}, {1,1}, {a,b,c}] // Simplify
Out[8]= { $\frac{3a - 2b + c}{4}, \frac{-a}{2} + b - \frac{c}{2}, \frac{a - 2b + 3c}{4}$ }
```

Our program can be compared with the function `TridiagonalSolve` in the standard package `LinearAlgebra`Tridiagonal``. Both functions use Gaussian elimination and take about the same time to execute. Neither function performs “pivoting,” so they can break down if a pivot becomes zero.

A Little Group Theory

As a last example, we look at the implementation of a finite group from a presentation. Consider the symmetry group of a square, that is, the dihedral group of order 4. The presentation has two generators x and y , and three relations. The generator x corresponds to a reflection, so we have the relation $x^2 = e$, where e is the identity element of the group. The generator y corresponds to the 90-degree rotation, so we have a second relation $y^4 = e$. The final relation is given by the interaction of the two generators: $yx = xy^3$. Using the built-in function `NonCommutativeMultiply` to represent the group multiplication, we can translate the relations into the following rules:

```
In[1]:= drules = {
  NonCommutativeMultiply[a___, e, b___] :=
    NonCommutativeMultiply[a, b],
  NonCommutativeMultiply[a___, e, b___] :=
    NonCommutativeMultiply[a, b],
  NonCommutativeMultiply[a___, x, x, b___] :=
    NonCommutativeMultiply[a, e, b],
  NonCommutativeMultiply[a___, y, y, y, y, b___] :=
    NonCommutativeMultiply[a, e, b],
  NonCommutativeMultiply[a___, y, x, b___] :=
    NonCommutativeMultiply[a, x, y, y, b],
  NonCommutativeMultiply[a_] := a};
```

The first two rules define the identity element. The next three rules give the relations of the presentation. Notice that we do not need to include rules to specify that multiplication is associative since `NonCommutativeMultiply` has the attribute `Flat`.

For dihedral groups of other orders, we need only change two of these rules, since the relations for the dihedral group of order n are $x^2 = e$, $y^n = e$, and $yx = xy^{(n-1)}$.

The rules can be used to simplify products of the generators:

```
In[2]:= x**y**x // . drules
Out[2]= y ** y ** y

In[3]:= y**x**y // . drules
Out[3]= x

In[4]:= y**x**y**y**x**y // . drules
Out[4]= e
```

We could not use this approach for all finite groups because some groups have an undecidable word problem, so the process of applying replacement rules is not guaranteed to terminate. However, the method works in this simple case and indeed we can take it further, giving a function to invert group elements.

```
In[5]:= invert[a_NonCommutativeMultiply] :=
  Reverse[a] /. y := y**y**y // . drules;
invert[a_] := a /. y := y**y**y
```

The two cases here take account of the possibilities that the element is a generator or a composition of generators. Here are a few examples of its use:

```
In[6]:= invert[y]
Out[6]= y ** y ** y

In[7]:= invert[y**y**x**y]
Out[7]= x ** y ** y ** y

In[8]:= x**y**y**invert[x**y**y] // . drules
Out[8]= e
```

The conjugation operation can be written as a function:

```
In[9]:= conj[a_, b_] := invert[b]**a**b // . drules
In[10]:= conj[x, y]
Out[10]= x ** y ** y


In[11]:= conj[y, x**y**y]
Out[11]= y ** y ** y
```

We could write other functions to check whether a set of group elements forms a subgroup or a normal subgroup, or a function that finds the normalizer of a set of elements (that is, the smallest normal subgroup which contains all the given elements). However, we will not spoil the fun that can be had in finding these for oneself.

References

Boolos, George S., and Richard C. Jeffery. 1974. *Computability and Logic*. Cambridge University Press.

J. M. Selig
School of Electrical, Electronic and Information Eng.,
South Bank University, Borough Road,
London SE1 0AA, U.K.

 The electronic supplement contains the notebook
Rule Based Programming.