

# *Algebraic Programming in Mathematica*

**Andrzej Kozłowski**

“Algebraic Programming” means programming based on the use of algebraic concepts, such as commutativity, associativity, and distributivity. It is a method of programming for which *Mathematica* is particularly well suited. We give several examples of algebraic programming in *Mathematica*, culminating with a solution of a problem that was posted to the *Math-Group* email list.

## ■ 1. What is Algebraic Programming?

I first came across the expression “the algebraic method in programming” (or algebraic programming for short) in the context of *Mathematica* in the book by Ilan Vardi [1]. By algebraic programming, Vardi meant solving programming problems by using concepts of abstract algebra, such as distributivity, commutativity, and associativity. Vardi gives some examples of algebraic programming. One of these is a solution to the following problem.

---

**Q:** Write a program that will produce all possible subsets of a given set (list) of symbols, for example  $\{a, b, c\}$ .

---

Vardi makes the observation that the answer to the problem is contained in the following algebraic formula.

```
In[1]:= s = (1 + a) (1 + b) (1 + c);
```

```
In[2]:= Expand[s]
```

```
Out[2]= 1 + a + b + a b + c + a c + b c + a b c
```

This formula depends only on the algebraic property of distributivity of multiplication and the existence of a unit. Indeed, *Mathematica* makes it possible to solve the problem directly by converting the last formula.

```
In[3]:= Variables /@ (Expand[s] /. Plus -> List) // Sort
```

```
Out[3]= {{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}
```

We can write a function that will apply this method to any set of symbols. Note, however, that this program will not work if we use a list of numbers!

```
In[4]:= subSets[s_List] :=
      Variables /@ (Expand[Times @@ (1 + s)] /. Plus -> List) // Sort
In[5]:= subSets[{a, b, c, d}]
Out[5]= {{}, {a}, {b}, {c}, {d}, {a, b}, {a, c}, {a, d}, {b, c}, {b, d},
      {c, d}, {a, b, c}, {a, b, d}, {a, c, d}, {b, c, d}, {a, b, c, d}}
```

This attractive method can be used to produce unexpected solutions to surprisingly many problems. However, this is not actually what Vardi meant by “algebraic programming.” In fact one might call this approach “naive algebraic programming.” It is naive because it makes use of specific algebraic functions rather than the general algebraic principles on which they are built. Here is a “sophisticated” algebraic program due to Vardi that achieves the same result.

```
In[6]:= subSetsV[s_List] := Distribute[{{}, {#}] & /@ s, List, List, List, Union]
In[7]:= subSetsV[{a, b, c, d}]
Out[7]= {{}, {d}, {c}, {c, d}, {b}, {b, d}, {b, c}, {b, c, d}, {a}, {a, d},
      {a, c}, {a, c, d}, {a, b}, {a, b, d}, {a, b, c}, {a, b, c, d}}
```

Not surprisingly, the second program not only works with symbols and numbers, but is also considerably faster than the first.

```
In[8]:= test = Array[a, {16}];
In[9]:= subSets[test]; // Timing
Out[9]= {9.37 Second, Null}
In[10]:= subSetsV[test]; // Timing
Out[10]= {3.13 Second, Null}
```

The main reason is that the second program is more “abstract,” in the sense that it makes fewer assumptions and has to perform fewer checks. Essentially it applies the principle of “distributivity” in the widest possible sense. In general, algebraic computations using functions like `Times`, `Expand`, and so forth involve extensive sorting of intermediate expressions before they are brought to a canonical form. From the point of view of efficiency, it is much better to operate with lists because no evaluation or canonicalization ever happens, and lists can contain anything.

## ■ 2. Coefficients of Polynomials and Series

Probably the best known use of algebraic programming is in solving combinatorial problems, where many solutions can be expressed in terms of so-called “generating functions.” Many examples are considered in [2]. Here I shall only briefly mention a problem that was sent to *MathGroup* in July 2001.

**Q:** Define a *Mathematica* function that counts the number of ways of partitioning a number into a fixed number of parts (repetitions allowed) taken from a given set.

Here is a solution that uses algebraic programming.

```
In[11]:= kpartitions[n_, k_, l_List] :=
SeriesCoefficient[Series[SeriesCoefficient[
Series[Times @@ ((1 / (1 - t*x^#)) & /@ l), {x, 0, n}], n],
{t, 0, k}], k]
```

Here is an example.

```
In[12]:= kpartitions[24, 5, {1, 2, 3, 4, 7, 11, 17, 19}]
Out[12]= 13
```

The proof of the correctness of this solution is left to the reader. The above solution, while elegant, suffers from the usual defects of naive algebraic programming; it is fairly slow. Much faster solutions can be found in the *MathGroup* archives.

### ■ 3. Attributes in *Mathematica*

Let us look at another example. Suppose we are given a list of  $n$  objects and want to construct all permutations of  $k$  objects from them. Again, we shall look for an algebraic program to realize this task. Let us consider an example where  $n = 4$ ,  $k = 3$ , and the objects are  $\{a, b, c, d\}$ . Then, as before, we see that we would get an answer by taking the terms of the expansion of

$$(a + b + c + d)(a + b + c + d)(a + b + c + d)$$

except that this time we do not want the multiplication to be commutative. *Mathematica* has the following function.

```
In[13]:= ? NonCommutativeMultiply
```

`a ** b ** c` is a general associative,  
but non-commutative, form of multiplication. More...

However it knows very few rules so if we are lazy we might just clear the `Orderless` attribute from `Times` and do something like this.

```
In[14]:= Perms[S_List, k_Integer] :=
Module[{perms}, ClearAttributes[Times, Orderless];
perms = Flatten /@ (Expand[(Plus @@ S)^3] /.
{Plus -> List, Times -> List, Power[x_, n_] -> Table[x, {n}]});
SetAttributes[Times, Orderless]; perms]
```

```
In[15]:= Perms[{a, b, c, d}, 3]
Out[15]= {{a, b, a}, {c, b, a}, {d, b, a}, {b, b, a}, {a, c, a},
          {b, c, a}, {d, c, a}, {c, c, a}, {a, d, a}, {b, d, a},
          {c, d, a}, {d, d, a}, {b, a, a}, {c, a, a}, {d, a, a}, {a, a, a},
          {b, a, b}, {c, a, b}, {d, a, b}, {a, a, b}, {a, c, b}, {b, c, b},
          {d, c, b}, {c, c, b}, {a, d, b}, {b, d, b}, {c, d, b}, {d, d, b},
          {a, b, b}, {c, b, b}, {d, b, b}, {b, b, b}, {b, a, c}, {c, a, c},
          {d, a, c}, {a, a, c}, {a, b, c}, {c, b, c}, {d, b, c}, {b, b, c},
          {a, d, c}, {b, d, c}, {c, d, c}, {d, d, c}, {a, c, c}, {b, c, c},
          {d, c, c}, {c, c, c}, {b, a, d}, {c, a, d}, {d, a, d}, {a, a, d},
          {a, b, d}, {c, b, d}, {d, b, d}, {b, b, d}, {a, c, d}, {b, c, d},
          {d, c, d}, {c, c, d}, {a, d, d}, {b, d, d}, {c, d, d}, {d, d, d}}
```

As before, this will not work for numbers instead of symbols. However, we could easily deal with this problem by mapping `Hold` on the list and then applying `ReleaseHold` at the end. Another problem is that we temporarily cleared the `Orderless` attribute of `Times`, which is generally not considered a very good idea and may produce unexpected results. In this case there were no such surprises. However, a general algebraic version of the code is both simpler and much more efficient.

```
In[16]:= Perms[S_List, k_Integer] := Distribute[Table[S, {k}], List]
In[17]:= Perms1[{a, b, c, d}, 3]
Out[17]= {{a, a, a}, {a, a, b}, {a, a, c}, {a, a, d}, {a, b, a},
          {a, b, b}, {a, b, c}, {a, b, d}, {a, c, a}, {a, c, b},
          {a, c, c}, {a, c, d}, {a, d, a}, {a, d, b}, {a, d, c}, {a, d, d},
          {b, a, a}, {b, a, b}, {b, a, c}, {b, a, d}, {b, b, a}, {b, b, b},
          {b, b, c}, {b, b, d}, {b, c, a}, {b, c, b}, {b, c, c}, {b, c, d},
          {b, d, a}, {b, d, b}, {b, d, c}, {b, d, d}, {c, a, a}, {c, a, b},
          {c, a, c}, {c, a, d}, {c, b, a}, {c, b, b}, {c, b, c}, {c, b, d},
          {c, c, a}, {c, c, b}, {c, c, c}, {c, c, d}, {c, d, a}, {c, d, b},
          {c, d, c}, {c, d, d}, {d, a, a}, {d, a, b}, {d, a, c}, {d, a, d},
          {d, b, a}, {d, b, b}, {d, b, c}, {d, b, d}, {d, c, a}, {d, c, b},
          {d, c, c}, {d, c, d}, {d, d, a}, {d, d, b}, {d, d, c}, {d, d, d}}
```

Here we see a characteristic difference between *Mathematica* and specialized programs for dealing with abstract algebraic structures like rings, groups, and so on. Such programs tend to be object oriented and contain structures like groups or rings as some form of data types. *Mathematica* does not have objects or types that correspond to groups or rings, but contains functions and attributes that let one implement distributivity, commutativity, associativity, and so forth in a functional way. As a bonus, we can find uses for them in more general programming that are not strictly concerned with abstract algebraic structures.

As we have seen, distributivity is implemented in *Mathematica* through the function `Distribute`. How about the other fundamental algebraic notions, commutativity and associativity? These are essentially implemented as a so-called “Attributes” of functions `Orderless` and `Flat`. Understanding these attributes (as well as `OneIdentity`, which does not have a simple algebraic motivation) is the foundation of algebraic programming.

In[18]:= ? Orderless

Orderless is an attribute that can be assigned to a symbol `f` to indicate that the elements `ei` in expressions of the form `f[e1, e2, ...]` should automatically be sorted into canonical order. This property is accounted for in pattern matching. More...

The attribute `Orderless` in a function is usually said to correspond to the mathematical notion of commutativity. This is indeed true, but one has to be careful with the interpretation of “correspond.” Many functions that are “mathematically” commutative do not have the attribute `Orderless`. Such is the case with the logical function `Or`. From the mathematical viewpoint the function is commutative.

In[19]:= << Experimental`

In[20]:= `ImpliesQ[Or[A, B], Or[B, A]]`

Out[20]= True

However, `Orderless` is not an attribute of `Or`.

In[21]:= `Attributes[Or]`

Out[21]= {Flat, HoldAll, OneIdentity, Protected}

We can see the effect this has on pattern matching.

In[22]:= `a || b /. _ || a -> c`

Out[22]= `a || b`

In[23]:= `b || a /. _ || a -> c`

Out[23]= `c`

The reason is not mathematical but has to do with performance and the way logical functions are used in programming, where one actually may wish to exploit the order of evaluation. The arguments of `Or[p, q, r, s, t, ...]` are evaluated in order, and `True` will be returned the moment one of them is found to be true. This may be a substantial timesaver. Also, consider the following example, which I owe to Daniel Lichtblau.

In[24]:= `cond1 := And[Length[a] ≥ 2, a[[2]] == 3];`  
`cond2 := And[a[[2]] == 3, Length[a] ≥ 2];`

We can see that the following two evaluations will produce different behavior.

In[26]:= `cond1 /. a -> {x}`  
`cond2 /. a -> {x}`

Out[26]= False

Part::partd : Part specification `a[[2]]` is longer than depth of object. More ...

Out[27]= False

In other words: we have to pay attention to order when pattern matching even though `||` is in fact a commutative operation. This can be exploited in programming both to increase efficiency and to avoid generating unnecessary errors.

On the other hand, it is precisely the effect of the “algebraic” attributes `Orderless`, `Flat`, and `OneIdentity` on pattern matching that makes algebraic programming a surprisingly powerful technique.

Let us then consider these remaining algebraic attributes, `Flat` and `OneIdentity`, which together with the function `Distribute` and the `Orderless` attribute, play the central roles in algebraic programming.

The attribute `Flat`, corresponds to the mathematical property of associativity. Of course, as usual there is more to it than just that.

```
In[28]:= ?Flat
```

`Flat` is an attribute that can be assigned to a symbol `f` to indicate that all expressions involving nested functions `f` should be flattened out. This property is accounted for in pattern matching. More...

In order to understand the effect of the algebraic attributes on pattern matching we shall use the *Mathematica* function `ReplaceList`. Indeed, this function, together with `Distribute` and the algebraic attributes, is the most useful *Mathematica* command in algebraic programming.

```
In[29]:= ?ReplaceList
```

`ReplaceList[expr, rules]` attempts to transform the entire expression `expr` by applying a rule or list of rules in all possible ways, and returns a list of the results obtained. `ReplaceList[expr, rules, n]` gives a list of at most `n` results. More...

Let us try using `ReplaceList` to understand the effect of the `Flat` attribute on pattern matching.

We start by taking a function `f` with no attributes and consider the following replacement rule.

```
In[30]:= ClearAll[f]
```

```
In[31]:= ReplaceList[f[a, b, c], f[a_, b_] -> g[a, b]]
```

```
Out[31]= {}
```

Not surprisingly there was no match. Now give `f` the attribute `Flat`.

```
In[32]:= SetAttributes[f, Flat]
```

Let us repeat the previous attempt.

```
In[33]:= ReplaceList[f[a, b, c], f[a_, b_] -> g[a, b]]
```

```
Out[33]= {g[f[a], f[b, c]], g[f[a, b], f[c]]}
```

Note what happened. The function  $f[a, b, c]$  was now rewritten using “associativity” in two ways, as  $f[f[a], f[b, c]]$  and  $f[f[a, b], f[c]]$ . The rule was

then applied but only at the top level. Now suppose we give  $f$  the attribute `OneIdentity` in addition to `Flat`.

```
In[34]:= SetAttributes[f, OneIdentity]
In[35]:= ReplaceList[f[a, b, c], f[a_, b_] -> g[a, b]]
Out[35]= {g[a, f[b, c]], g[f[a, b], c]}
```

You see that all expressions of the type  $f[x]$  were replaced just by  $x$ . Of course this happens only when pattern matching. If you enter

```
In[36]:= f[x]
Out[36]= f[x]
```

you still get just  $f[x]$ , not  $x$ .

Indeed:

```
In[37]:= ? OneIdentity
```

`OneIdentity` is an attribute that can be assigned to a symbol  $f$  to indicate that  $f[x]$ ,  $f[f[x]]$ , etc. are all equivalent to  $x$  for the purpose of pattern matching. More...

The useful function `ReplaceAllList` works like `ReplaceList` but also works on all subexpressions.

```
In[38]:= ReplaceAllList[expr_, rules_] :=
Module[{i}, Join[ReplaceList[expr, rules],
If[AtomQ[expr], {}, Join@@Table[ReplacePart[expr, #, i] & /@
ReplaceAllList[expr[[i]], rules], {i, Length[expr]}]]]]
```

Let us now look at another example of algebraic programming. Of course, the most natural use of algebraic programming is solving algebra problems. So let us first consider an associativity problem.

**Q:** Suppose  $g$  is a nonassociative noncommutative product of any number of elements. Find all the distinct ways of multiplying four elements using  $g$  that would be equal if  $g$  were associative.

We use a function  $f$ , to which we assign the attributes `Flat` and `OneIdentity`.

```
In[39]:= ClearAll[f]; SetAttributes[f, {Flat, OneIdentity}]
In[40]:= FixedPoint[Union[Flatten[ReplaceAllList[#, f[a_, b_] -> g[a, b]]] &,
f[a, b, c, d]]
Out[40]= {g[a, f[b, c, d]], g[a, g[b, f[c, d]]],
g[a, g[b, g[c, d]]], g[a, g[f[b, c], d]], g[a, g[g[b, c], d]],
g[f[a, b], f[c, d]], g[f[a, b], g[c, d]], g[f[a, b, c], d],
g[g[a, b], f[c, d]], g[g[a, b], g[c, d]], g[g[a, f[b, c]], d],
g[g[a, g[b, c]], d], g[g[f[a, b], c], d], g[g[g[a, b], c], d]}
```

```
In[41]:= Union[% /. f -> g]
```

```
Out[41]= {g[a, g[b, g[c, d]]], g[a, g[g[b, c], d]],
          g[a, g[b, c, d]], g[g[a, b], g[c, d]],
          g[g[a, g[b, c]], d], g[g[g[a, b], c], d], g[g[a, b, c], d]}
```

This sort of thing is in fact quite useful in various areas of mathematics. Later we shall give just one application of exactly this idea to solving a rather well-known puzzle that is frequently asked on symbolic algebra email lists.

## ■ 4. The 4 Fours Problem

The following problem was sent to the *MathGroup* email list in June 2002.

---

**Q:** Find the numbers 0 to 99 using any rational operator (i.e., an operation which results in a rational number) and 4 fours, where each equation must contain no more and no less than 4 fours. Not all numbers may be possible. Some examples include:

$$0 = 4 + 4 - 4 - 4$$

$$1 = 4/4 + (4 - 4)$$

$$2 = 4/4 + 4/4$$

$$27 = 4! + 4/4 + \sqrt{4}.$$

---

The problem as formulated is rather unclear because of the vagueness of the phrase “rational operator.” The question allowed not only binary operations but also unary operations, such as taking the square root or factorials. This of course means that there is an infinite number of possible solutions unless one imposes additional restrictions. However, we shall first simplify the problem and consider only binary operations: addition, subtraction, multiplication, division, and exponentiation. Let us try to solve the following simpler version of the problem.

---

**Q:** Find all integers between 1 and 99 that are expressible using 4 fours and only the arithmetical operations: +, -, \*, /, and ^.

---

This is very easy to implement in *Mathematica* using algebraic programming. In fact, we shall use exactly the same method as in the last example.

The idea is very simple. First we find all ways of writing expressions like  $A_0[A_1[4, 4], A_2[4, 4]]$ , where the  $A_i$  are the binary operations *Times*, *Plus*, *Divide*, *Subtract*, and *Power*. Actually in *Mathematica*, *Times* and *Plus* are  $n$ -ary operations for any  $n \geq 1$ , but *Divide*, *Subtract*, and *Power* are not. To handle them all simultaneously, we consider them to be binary operations.

It is clear that the number of different expressions that can be made up of four arguments and binary operations  $A_0, A_1, \dots$  (which may be different or the same)

can be described as follows. Let  $A$  be an associative binary operation. Then expressions such as  $A[x, A[y, A[z, w]]]$  and  $A[A[x, y], A[z, w]]$  are equal. Clearly the number of such expressions that are equal by virtue of the associativity of  $A$  (which we found in the previous section) is equal to the number of distinct ways of applying three binary functions to four arguments. In fact we can obtain all the expressions of the latter type from those of the former as follows.

First we repeat what we did above, that is, define an associative operation  $f$ .

```
ClearAll[f];
SetAttributes[f, {Flat, OneIdentity}]
```

Next, just as above, we find all the ways of “associating” four elements.

```
In[43]:= ls = Union[
  FixedPoint[Union[Flatten[ReplaceAllList[#, f[a_, b_] -> g[a, b]]]] &,
    f[a, b, c, d]] /. f -> g]
Out[43]= {g[a, g[b, g[c, d]]], g[a, g[g[b, c], d]],
  g[a, g[b, c, d]], g[g[a, b], g[c, d]],
  g[g[a, g[b, c]], d], g[g[g[a, b], c], d], g[g[a, b, c], d]}
```

Then we remove all duplicates and expressions that involve the application of  $g$  to three elements.

```
In[44]:= ls = DeleteCases[ls, _?(Not[FreeQ[#, g[x_ /; Length[{x}] ≥ 3]]] &), 1]
Out[44]= {g[a, g[b, g[c, d]]], g[a, g[g[b, c], d]],
  g[g[a, b], g[c, d]], g[g[a, g[b, c]], d], g[g[g[a, b], c], d]}
```

Finally we replace  $g$  by  $A_2, A_1, A_0$ , in this order, and make all the elements  $\{a, b, c, d\}$  equal to 4.

```
In[45]:= ls =
  (i = 0; MapAll[If[# === g, # /. g -> Subscript[A, Mod[++i, 3]], #] &, ls,
    Heads -> True] /. Thread[Rule[{a, b, c, d}, 4]])
Out[45]= {A1[4, A2[4, A0[4, 4]]], A1[4, A2[A0[4, 4], 4]],
  A1[A2[4, 4], A0[4, 4]], A1[A2[4, A0[4, 4]], 4], A1[A2[A0[4, 4], 4], 4]}
```

(A different approach to the same problem can be found in exercise 13 of Chapter 6 of [2].)

We shall now simply substitute the five binary operations Plus, Subtract, Times, Divide, and Power for  $A_0, A_1, A_2$  in all possible ways. First we need to redefine Divide and Power. Basically we do this to avoid error messages caused by division by 0. For later use, we also place a restriction on the maximum and minimum size of the exponent to avoid unnecessary computations of very large and very small numbers.

```
In[46]:= Off[General::"spell1"]; Off[General::"spell"];
power[x_Integer?(# > 0 &), y_Integer?(-16 ≤ # ≤ 16 &)] := x^y;
power[x_] := Indeterminate;
divide[x_, y_] /; y ≠ 0 := Divide[x, y]; divide[x_, 0] := Indeterminate;
```

Next we create a set of rules that we shall use to create all possible ways of replacing the  $A_i$  with the binary operations.

```
In[49]:= rules =
  Map[Thread, Map[RuleDelayed[Array[A#-1 &, {5}], #] &, Distribute[
    Array[{Times, Plus, Subtract, divide, power} &, {5}], List]]];
```

Here is what the list we created looks like.

```
In[50]:= Take[rules, 5]
Out[50]:= {{A0 -> Times, A1 -> Times, A2 -> Times, A3 -> Times, A4 -> Times},
  {A0 -> Times, A1 -> Times, A2 -> Times, A3 -> Times, A4 -> Plus},
  {A0 -> Times, A1 -> Times, A2 -> Times, A3 -> Times, A4 -> Subtract},
  {A0 -> Times, A1 -> Times, A2 -> Times, A3 -> Times, A4 -> divide},
  {A0 -> Times, A1 -> Times, A2 -> Times, A3 -> Times, A4 -> power}}
```

In fact, since an actual formula never involves more than three binary operations, we remove the rules involving unnecessary  $A_s$ .

```
In[51]:= rules = Union[DeleteCases[rules,
  HoldPattern[A3 -> _] | HoldPattern[A4 -> _], Infinity]];
In[52]:= Take[rules, 5]
```

```
Out[52]:= {{A0 -> divide, A1 -> divide, A2 -> divide},
  {A0 -> divide, A1 -> divide, A2 -> Plus},
  {A0 -> divide, A1 -> divide, A2 -> power},
  {A0 -> divide, A1 -> divide, A2 -> Subtract},
  {A0 -> divide, A1 -> divide, A2 -> Times}}
```

Let us now find all possible solutions of the “easy” 4 fours problem.

```
In[53]:= solutions =
  Union[Flatten[Union[Union[Select[Flatten[Union[# /. rules]],
    Element[#, Integers] && 0 <= # <= 99 &]] & /@ 1s]]]
Out[53]:= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 15, 16, 17,
  20, 24, 28, 32, 36, 48, 60, 63, 64, 65, 68, 80, 81}
In[54]:= Length[solutions]
Out[54]:= 27
```

Finally let us find all possible ways of obtaining a given number, for example, 7.

```
In[55]:= patterns =
  Union[Flatten[Union[Flatten[Union[HoldForm[#] /. rules]]] & /@ 1s]];
In[56]:= Select[patterns, ReleaseHold[#] == 7 &] /.
  {divide -> Divide, power -> Power}
Out[56]:= {(4 -  $\frac{4}{4}$ ) + 4, 4 + (4 -  $\frac{4}{4}$ ), 4 - ( $\frac{4}{4}$  - 4), (4 + 4) -  $\frac{4}{4}$ }
```

This completely solves the “easy” 4 fours problem. We can solve the analogous 5 fives problem and so on in exactly the same way. In the final section we take a look at the “hard” 4 fours problem presented in the original question.

## ■ 5. The Hard 4 Fours Problem

The hard version of the problem is to find all possible representations of all numbers between 1 and 99 using any “reasonable” operations, including unary ones (“functions”) like `Sqrt` or `Factorial`. Of course there is now an infinite number of possible ways of trying to write out a solution, even with a finite number of operations. We shall therefore concentrate only on representations using `Sqrt` and `Factorial` in addition to the usual arithmetic operations, and only try to find some additional solutions. (If the additional operations were binary [e.g., `Max` or `Min`] then we could use exactly the same method as before to obtain a complete solution.) We first introduce modified definitions of the additional operations.

```
In[57]:= sqrt[x_? (# > 0 &)] := Sqrt[x]; sqrt[x_] := Indeterminate;
factorial[n_Integer? (0 <= # < 12 &)] = Factorial[n];
factorial[n_] := Indeterminate;
```

We create a new set of rules that transform our unary functions  $B_i$  into `factorial`, `sqrt`, or `Identity` to account for the cases where no unary operation is used.

```
In[59]:= rules1 = Map[Thread, Map[RuleDelayed[Array[B_ &, {4}], #] &, Distribute[
Array[{Identity, Identity, factorial, sqrt} &, {4}], List]]];
```

```
In[60]:= Take[rules1, 5]
```

```
Out[60]= {{B1 -> Identity, B2 -> Identity, B3 -> Identity, B4 -> Identity},
{B1 -> Identity, B2 -> Identity, B3 -> Identity, B4 -> Identity},
{B1 -> Identity, B2 -> Identity, B3 -> Identity, B4 -> factorial},
{B1 -> Identity, B2 -> Identity, B3 -> Identity, B4 -> sqrt},
{B1 -> Identity, B2 -> Identity, B3 -> Identity, B4 -> Identity}}
```

We see that there are rather a lot of identical rules. We can reduce their number.

```
In[61]:= rules1 = Union[rules1, SameTest ->
(((Array[B_ &, {4}] /. #1) === (Array[B_ &, {4}] /. #2)) &)];
```

```
In[62]:= Take[rules1, 5]
```

```
Out[62]= {{B1 -> factorial, B2 -> factorial, B3 -> factorial, B4 -> factorial},
{B1 -> factorial, B2 -> factorial, B3 -> factorial, B4 -> Identity},
{B1 -> factorial, B2 -> factorial, B3 -> factorial, B4 -> sqrt},
{B1 -> factorial, B2 -> factorial, B3 -> Identity, B4 -> factorial},
{B1 -> factorial, B2 -> factorial, B3 -> Identity, B4 -> Identity}}
```

The most obvious expressions to evaluate are obtained from the earlier ones by replacing all fours by  $B_i[4]$ .

```
In[63]:= expr[j_] :=
Module[{i = 0}, MapAll[If[# === 4, Subscript[B, ++i][#], #] &, ls[[j]]]]
```

Here are all the solutions we get in this way.

```
In[64]:= extendedsols = Union[Flatten[
      Union[Select[Flatten[Union[Union[expr[#] /. rules] /. rules1]],
        Element[#, Integers] && 0 < # <= 99 &]] & /@ Range[5]]]
Out[64]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
      21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 34, 35, 36, 37, 38,
      40, 42, 44, 46, 47, 48, 49, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68,
      70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 95, 96, 97, 98}
```

```
In[65]:= Length[%]
Out[65]= 72
```

An interesting case is the number 31, which was not among the solutions we got earlier. Let us see how it is represented.

```
In[66]:= extendedpatterns = Union[Flatten[
      Union[Union[HoldForm[Evaluate[expr[#]]] /. rules] /. rules1] & /@
      Range[5]]];
In[67]:= Select[Flatten[extendedpatterns], ReleaseHold[#] == 31 &] /.
      {divide -> Divide, power -> Power, sqrt -> Sqrt,
      factorial -> Factorial} /. HoldPattern[Identity[x_]] -> x
Out[67]= {4! +  $\frac{4+4!}{4}$ , 4! +  $\frac{4!+4}{4}$ ,  $\frac{4+4!}{4} + 4!$ ,  $\frac{4!+4}{4} + 4!$ }
```

Of course there are in principle infinitely many ways to insert more  $B$ s into different slots, and it is not clear how many more numbers can be obtained in this way. Here is just one example. Consider the pattern  $A_0[A_2[B_2[A_1[B_1[4], 4]]]$ ,  $B_3[4]$ ,  $B_4[4]$ .

```
In[68]:= extrasols = Union[Select[Flatten[Union[
      Union[A_0[A_2[B_2[A_1[B_1[4], 4]]], B_3[4]], B_4[4]] /. rules] /. rules1]],
      Element[#, Integers] && 0 < # <= 99 &]]
Out[68]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18,
      20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 34, 36, 38,
      40, 42, 44, 45, 46, 47, 48, 49, 50, 52, 54, 56, 58, 60, 62, 64,
      66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 88, 90, 92, 94, 96, 98}
```

We see that it produced just one new number.

```
In[69]:= Complement[extrasols, extendedsols]
Out[69]= {45}
```

However, we created several new representations for the solutions already found. Here is another representation of 31.

```
In[70]:= extrapatterns = Flatten[Union[
      Union[HoldForm[A_0[A_2[B_2[A_1[B_1[4], 4]]], B_3[4]], B_4[4]]] /. rules] /.
      rules1];
In[71]:= Select[extrapatterns, ReleaseHold[#] == 31 &] /.
      {divide -> Divide, power -> Power, sqrt -> Sqrt,
      factorial -> Factorial} /. HoldPattern[Identity[x_]] -> x
Out[71]= { $\frac{\frac{4!}{4}! + 4!}{4!}$ ,  $\frac{(\sqrt{4} + 4)! + 4!}{4!}$ ,  $\frac{4! + 4}{4} + 4!$ }
```

Here are all the solutions we have found so far.

```
In[72]:= solutions = Union[extendedsols, extrasols]
```

```
Out[72]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,  
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 34, 35, 36, 37, 38, 40,  
42, 44, 45, 46, 47, 48, 49, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68,  
70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 95, 96, 97, 98}
```

Here are the remaining outstanding numbers.

```
In[73]:= Complement[Range[99], solutions]
```

```
Out[73]= {33, 39, 41, 43, 51, 53, 55, 57, 59, 61, 63, 65, 67,  
69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 99}
```

See if you can find interesting representations of these, possibly using some other operations.

## ■ Acknowledgments

I would like thank Daniel Lichtblau and Michael Trott [3] for useful comments and suggestions.

## ■ References

- [1] Ilan Vardi, *Computational Recreations in Mathematica*, Reading, MA: Addison-Wesley, 1991.
- [2] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*, Reading, MA: Addison-Wesley, 1989.
- [3] Michael Trott, *The Mathematica Guidebook to Programming*, Berlin: Springer-Verlag, 2003.

## About the Author

Andrzej Kozłowski was a professor of mathematics at Toyama International University in Japan when he wrote this article. He is now teaching mathematics and physics at Tokyo Denki University in Japan. Most of his research has been in algebraic topology, but recently he has become interested in mathematical finance, which he is teaching as an Internet-based course at Warsaw University. He recently joined Chatham Research ([www.chathamresearch.com](http://www.chathamresearch.com)) in a project to apply *Mathematica* to problems in quantitative finance and economics. He is a frequent contributor to the *MathGroup* email list.

**Andrzej Kozłowski**  
Tokyo Denki University  
Faculty of Information Environment  
Chiba, Japan  
[akoz@mimuw.edu.pl](mailto:akoz@mimuw.edu.pl)