Reprint from
The *Mathematica* Conference
June, 1992   Boston, MA

856.15.10.1992

Mathematica Programming Style

John M. Novak

The issue of "style" when related to *Mathematica* programming is a multifaceted concept. We have on the one hand the idea of coding style: what programming structures are most efficient, what structures are most elegant. But programming is more than this, particularly in an interpreted environment such as *Mathematica*. The ideas of elegant and efficient code apply equally well to code fragments and small routines as well as large systems.

However, in *Mathematica*, there are elements of style associated with large systems, as well. The usual approach is to place a series of related routines, or a single large algorithm, into a *package*. The package not only collects these routines, but contains ancilliary information such as usage messages, error messages, and variable localization constructs. While these issues are often addressed under the heading "program design", I believe that in an environment such as *Mathematica*, these are intrinsically stylistic elements.

The following text will address all of these issues, drawing upon existing packages in several instances.

1 Efficient Code

Mathematica is at heart a functional programming language. However, the syntax of many of its programming constructs is very similar to those of traditional procedural languages. It has powerful pattern matching capabilities, and the ability to create a database of rules to transform expressions. It has a very rich collection of mathematical functions. Together, this information can tell us a bit about which structures are most effective for a given purpose.

Consider, for example, the traditional first program in many languages—a function to calculate the factorial of a positive integer. Because of the richness of *Mathematica*'s syntax, we can code this basic example in many ways (see below).

```
factorialProcedural[x_] :=
        Module[{cumulative,index},
                If[Not[IntegerQ[x] && PositiveQ[x]],
                        Return["Bad Input"]];
                cumulative = 1; index = 1;
                While[index <= x,
                        cumulative = cumulative * index;
                        index = index + 1;
                ];
                Return[cumulative]
        ]

factorialFunctional[x_] :=
        If[x <= 1,
                Return[ 1 ],
                Return[ x factorialFunctional[x - 1]] ]

factorialRuleBased[1] = 1;
factorialRuleBased[x_Integer?Positive] :=
                x factorialRuleBased[x - 1]

factorialFunctional2[x_Integer?Positive] :=
        Times @@ Range[x]
```

Different ways of programming the factorial.

We can compare the times of execution of these various methods, running each version over a range of values. This can give us a rough idea of the effectiveness of each method.

factorialProcedural	50 seconds
factorialFunctional	55 seconds
factorialRuleBased	45 seconds
factorialFunctional2	6 seconds

Timings of various factorial implementations, for first hundred positive integers, on Macintosh IIcx.

We see that a non-traditional syntax proves most effective. Taking advantage of the ability to modify the head of an expression can lead to enormous improvements in performance. The importance of the rule-based concept is also evident; not only do we have faster execution, but we can easily code more robust error checking than traditional procedural techniques with minimal difficulty. See Roman Maeder's article in April 92 *Dr. Dobb's Journal*, and his tutorials on *Mathematica* programming, for more information on these issues.

It is instructive to compare our above functions to the built-in factorial function. The same calculation of the factorials of the first hundred integers took about .6 seconds. The lesson is clear; above all, use internal functions when possible. In the vast majority of cases, this will improve execution time tremendously.

The nature of the *Mathematica* environment can subtly influence performance in other ways. Consider, for instance, the way symbols are handled. All symbols are effectively global; localization in performed by special naming conventions, such as contexts. There are a number of routines and behaviors of the *Mathematica* environment to enforce this localization; however, access of a symbol in any context is possible from any other context if the full name of the symbol is used. This global nature causes interesting effects such as decreased performance when large numbers of symbols are present; most users will not see this effect, but it is something to be aware of in very large systems.

Another result of this behavior is performance differences between the localization constructs `Block` and `Module`. `Block` only maintains a local value for a symbol; `Module` maintains a local instance of a symbol. This leads to an obvious performance hit; while `Block` only has to cache the current value of the symbol, `Module` must declare and add to the symbol table an entire new instance of the symbol. However, `Module` is much more robust in its localization of a symbol; thus it is the preferred construct. `Block` should be used mostly in the instance where a small routine is to be called a large number of times, when the performance difference becomes noticeable.

Mathematica's unique syntax, in which assignment is performed by an expression, and rules for various inputs to a function can be easily established, allows for a useful technique, especially for recursive methods. This is caching, where the initial calculation for a particular input is retained, so that further calls to perform the calculation return the result directly. For instance, following is a simple example involving the calculation of values of the Fibonacci sequence.

```
fibonacci[0] = 1; fibonacci[1] = 1;
fibonacci[n_Integer?Positive] :=
        fibonacci[n - 1] + fibonacci[n - 2]

fibonacci2[0] = 1; fibonacci2[1] = 1;
fibonacci2[n_Integer?Positive] :=
        fibonacci2[n] =
        fibonacci2[n - 1] + fibonacci2[n - 2]
```

A Fibonacci algorithm, without and with caching of intermediate results.

Because this algorithm makes many repetitive calls to calculate the function for the same inputs, it becomes useful to cache those values. Examine the improvement in performace between the two versions: calculating a table of the first 15 numbers in the sequence on a Macintosh IIcx takes 39 seconds without caching; but only 0.8 seconds with caching.

The final technique of this class that we will investigate is of less general applicability. The use of the `Compile` function is only applicable to functions that are to perform machine-precision floating-point calculations, and contain only the simpler (mostly procedural) programming constructs. Other kinds of calculations can be run through the compiler without harm, but this will only degrade performance due to handling the unnecessary compiler information. However, for the above class of computations, improvements of up to 20 times can be seen. Here is such an example, involving the calculation of the Mandelbrot set.

```
mandelbrot[x_,y_,lim_] :=
        Module[{z,ct = 0},
                z = x + I y;
                While[Abs[z] < 2.0 && ct <= lim,
                        z = z^2 + (x + I y);
                        ++ct
                ];
                ct
        ]

mandelbrotC =
Compile[{x,y,lim},
        Module[{z,ct = 0},
                z = x + I y;
                While[Abs[z] < 2.0 && ct <= lim,
                        z = z^2 + (x + I y);
                        ++ct
                ];
                ct
        ]];

DensityPlot[-mandelbrotC[x,y,50],
        {x,-2.,1.},{y,-1.5,1.5},
                PlotPoints->50,Mesh->False]
```

Determining whether a point is in the Mandelbrot set: uncompiled vs. compiled version, followed by a command that can be used to display the set.

The compiler works by converting the *Mathematica* code into a generic machine pseudo-code for a stack-based machine. Calls to internal functions are made directly. This simpler command set can be evaluated substantially faster than the usual *Mathematica* command set, although it is still interpreted.

These are but a few of the issues and techniques to consider when creating efficient code; I would recommend reading *The Mathematica Journal* as a source of other methods to keep in mind.

2 Elegant Code

"Elegance" in code is often a rather insubstantial concept; it can mean the most compact, or the clearest, or the cleverest, or combinations of these. With *Mathematica*'s rich syntax, creating elegant code is something of an art form. I will not attempt to explore this to any depth; rather, I would recommend reading Roman Maeder's book *Programming in Mathematica*, or certain parts of Ilan Vardi's book *Mathematica Recreations*. However, there are some very useful, but little understood, constructs that can assist immensely in the creation of elegant code.

The first of these is the `Fold` function. The arguments are a pure function, an initial expression, and a list of additional expressions. The pure function should be able to take two arguments. The result of `Fold`, then, is the application of the function to the previous recursive iteration of `Fold` and the last element of the list, where the first iteration is the application of the function to the initial expression and the first element of the list. To make this clearer, consider the following equivalent procedural function.

```
myfold[func_, initial_, inputlist_] :=
    Module[{tmp = initial, list = inputlist},
        While[list != {},
            tmp = func[tmp, First[list]];
            list = Drop[list,1]
        ];
        tmp
```

```
    ]
```

This can lead to very elegant code; consider, for instance, the following code to orthogonalize a set of vectors by the Gram-Schmidt method. Note that an extended version of this, with methods for normalizing the projection, and orthogonalizing across an arbitrary inner product (such as a function space) is implemented in the package `LinearAlgebra`Orthogonalization``.

```
(* Project one vector on another *)
Projection[v1_,v2_] := (v1 . v2) v2/(v2 . v2)

(* Project across a list of vectors *)
MultipleProjection[v1_, vecs_] :=
        Plus @@ Map[Projection[v1, #]&, vecs]

(* Orthogonalize list of vectors (matrix) *)
GramSchmidt[mat_] :=
        Fold[Join[#1, {#2 - MultipleProjection[#2,#1]}]&, {}, mat]
```

Another particularly useful construct is the generalized outer product. Although this originated as a math function, for dealing with tensors, it can be applied to generic expression, and is a useful way of combining lists. For instance, where one has two lists, and requires all possible combinations, `Outer` can be used to return an array of those combinations. The syntax is `Outer[`*function*`,` *list1*`,` *list2*`]`. The value returned would be an array of *function* applied to each pair of elements of *list1* and *list2*.

The classic example of this is the Jacobian matrix.

```
Jacobian[funs_List, vars_List] := Outer[D, funs, vars]
```

Use of `Outer` to produce the Jacobian matrix.

Finally, an important warning: do not let the desire for elegant and efficient code override the need for robust code. Consider, for instance, the code for generating continued fraction representations (see The *Mathematica* Book, page 12) compared to the current package `NumberTheory`ContinuedFractions``. You will note that it has taken on a somewhat less elegant form, but one able to handle a wider range of conditions, and properly react to invalid conditions that the more concise version fails on.

3 Package Design

Many of the stylistic issues associated with package design are simply those of good programming practice. Code that is properly commented is much more useful to other people than code that is not. Appropriate modularization of code allows pieces to be easily reused in other packages, as well as simplification of the package structure, thereby reducing redundant code. (In fact, it is sometimes appropriate to create an entirely new package from a particularly useful routine, so that other packages can simply require its presence by use of the `Needs` command. A good example of this is the package `Utilities`FilterOptions``; a good half of my packages use this for processing options.)

However, there are other issues that are a bit more specific to *Mathematica*. These are more generally expressed as constraints on style. They usually relate to integrating your own code with the *Mathematica* interface.

First, let's consider the traditional structure of a *Mathematica* package. A package usually begins with assorted identifying information (see the submission guidelines for *The Mathematica Journal* for a standard for this information.) A package context is then declared by the `BeginPackage` construct. Note that this can also load any ancilliary packages required by this package. The purpose of declaring a context for each package is so other packages can be loaded with symbols of the same name, and the definitions will not conflict. (If one were to load all of the standard *Mathematica* packages, one would see the necessity of this.) Following the package declaration, the symbols visible in this context are usually declared via the establishment of usage messages for each one. Next, a private context is declared, under the package context. Symbols in this context are not normally externally visible (the context is not placed on the context path); this is a hiding feature that also helps prevent naming conflicts. The functions, along with accompanying error messages and option declarations are given, finally followed by functions that end the private context and the package.

Stylistic design issues apply to several parts of the above description. For instance, it is generally not a good idea to declare more than one context per a given package. One reason for this is that it would make use of the `Master`package` mechanism rather difficult. Master packages are a series of `DeclarePackage` commands; these effectively declare a number of symbols in a given context, without loading the package that actually defines that symbol. When the symbol is then called upon, the package is loaded. It is loaded from a file whose name corresponds to the context given in the `DeclarePackage` statement. It can be useful to have several master packages loaded in one's `init.m` file, to help prevent the classic syndrome of trying to use a function before the package is loaded.

```
DeclarePackage["Graphics`FilledPlot`",
  {"Axis", "Back", "Curves", "FilledPlot", "Fills", "Front"}]
```

A package declaration from `Graphics`Master``.

Usage messages are another place where care should be taken. First, it is a good idea to only make externally visible those functions that are truly useful; internal calls can be examined by the full context name. If made externally visible, they can increase the possibility of name shadowing by loading other packages that have similiarly named functions. The phrasing of the message is itself important; the usage message should always begin with a syntax declaration. Compare the following messages.

```
WorldPlot::usage = "WorldPlot is a  function to generate maps
        of the world which draw data from a specified data base.
        WorldPlot[list], where list is a list of countries in the
        data base, produces a map of those countries.
        WorldPlot[{list,shading}] produces a map where shading is
        either a function applied to each name in list, producing
        a color or grayscale, or is a list of shades, one per name
        in list. A Graphics object or a WorldGraphics object can
        be produced.  Accepts WorldGraphics and Graphics options.";

WorldPlot::usage =
"WorldPlot[list] is a  function to generate maps of the world which
draw data from a specified data base, where list is a list of countries
from the data base.  Country names are usually specified as strings.
WorldPlot[{list,shading}] produces a map where shading is either a
function applied to each name in list, producing a color or grayscale,
or is a list of shades, one per name in list. A Graphics object or a
WorldGraphics object can be produced.  Accepts WorldGraphics and
Graphics options.";
```

Compare incorrect and correct versions of a usage message.

The reason for this is that the Notebook front end template mechanism examines the beginning of the usage message for the template to return. Thus, for the front end to be able to use your usage message, it needs the template at the start of the message.

Error messages also need to be properly presented to the user. Techniques often used include using print statements when the error occurs, and returning an error condition as the output of the function. Neither of these is a truly appropriate mechanism. Rather, the standard message system should be used. One reason for this is if your routines are called from another package; it is easier to localize the origin of a message, and to correctly handle error conditions, if an error message is generated. For instance, `Check` can be used to watch for the generation of particular messages; there is no way to watch for the emission of a `Print` statement.

It is also worthwhile to examine the naming and syntax of one's functions, particularly if they are intended for use by other people. Functions should be named in the same fashion as similar kernel functions. This adds a mnemonic effect that simplifies use of the functions. One of the philisophical design principles of *Mathematica* is to attempt to use as little cryptic notation as possible. Compare a program in *Mathematica* with a program in another language, and the one in *Mathematica* will almost invariably be easier to read. Your own naming conventions should attempt to mimic this wherever possible. Syntax, too, should be similar to existing functions; for instance, if your function reads in data, look to the `Read` function for precedence. The first argument is a stream, the second a controlling factor, followed by options. This gives you a model to follow, so that users will find it easier to adapt to your functions. For instance, examine the syntax of the package `` Utilities`BinaryFiles` ``; where it is appropriate, the syntax of the standard functions is followed, but where special considerations come to light, the syntax is modified.

The use of options should also follow this pattern. Whenever information needs to be specified to modify the form of the output of the function, or to modify the behavior of the algorithm, options in the form of rules are used. The advantage of this syntax is that defaults are easily set, and swiftly modifiable by the user. Also, it is easy to extend the syntax without interfering with the order of existing arguments. As an example of this principle, look to the kernel function `Solve`. Essential information is given as arguments, which are sometimes optional where that information can be determined from other input (*e.g.*, variable names). Information that modifies the nature of the algorithm, however, is given as options; the defaults can then be modified by the user.

4 Conclusions

We have examined various issues related to coding efficient and elegant programs in *Mathematica*. We have also examined some issues to consider when creating *Mathematica* packages. Application of these ideas will help other *Mathematica* users appreciate your programs, by virtue of their efficiency and standardization.