

# *Leibniz: Implementing a Drag-and-Drop Calculator*

**Joe Gregg**

*Lawrence University  
Appleton, WI 54911  
[greggi@lawrence.edu](mailto:greggi@lawrence.edu)*

**Leibniz is a combination word processor and front end for *Mathematica* designed for use in mathematics education. Leibniz uses a sophisticated pattern matching scheme to implement a simple and elegant interface for performing symbolic calculations. This article will describe the architecture of the Leibniz system and discuss some of the special techniques used in Leibniz to control evaluation.**

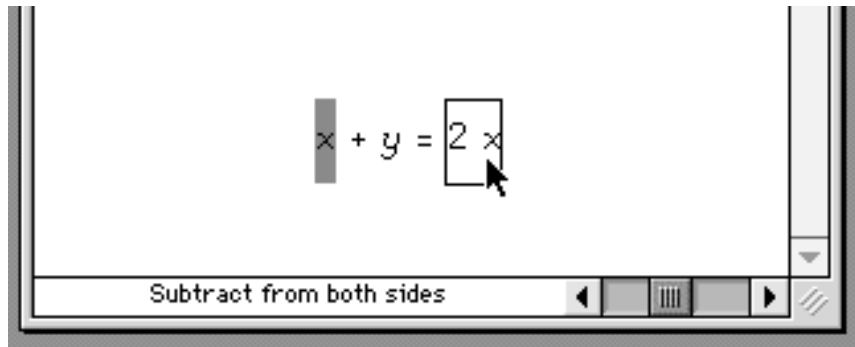
## ■ What is Leibniz?

Leibniz is a stand-alone text and equation editor for Windows and MacOS computers implemented in C++. Leibniz communicates with the *Mathematica* kernel via *MathLink*, sending requests for calculations to the kernel and formatting and displaying the results. In addition to the application itself, the system also includes **LZCompute.m**, a *Mathematica* package that mediates the communication between the Leibniz front end and the kernel.

Leibniz implements a powerful and easy-to-use mouse-driven calculation system. Users perform calculations by selecting a portion of an expression to evaluate and pressing enter, or rearrange the expression by dragging a selection from one part of an expression to a new location. For example, to compute an integral the user would type the integral, select it, and press enter. To move a term from one side of an equation to the other by, say, subtracting the term from both sides, the user selects the term and drags it to the other side of the equation. This system is particularly powerful and easy to use because Leibniz takes advantage of *Mathematica*'s powerful pattern matching capabilities to match the form of an expression to manipulations that could be performed on it.

When the user selects a portion of an equation, the application parses the equation and translates it into an internal data structure that reflects the structure of the equation. At the same time, the application calls a function contained in the **LZCompute** package. The purpose of this function is to examine the structure of the expression the user has selected and to return a list of possible mathematical manipulations that could be performed on the expression. These possible manipulations come back ranked in order of likelihood, with the most likely option displayed to the user in a message panel located in the lower left corner of the document window. The user can then press enter to have that calculation carried out, or hold down the command key and click on the selection to pop up a menu containing the other calculation options contained in the list.

Dragging operations are implemented in a similar fashion. As soon as the user clicks in the selected portion of an expression and starts to move the mouse, Leibniz queries the kernel to see if any manipulations are possible. As the user drags the mouse into each new part of an expression, the Leibniz front end calls a function in the **LZCompute** package to determine whether or not that portion of the expression is a valid target for a drag-and-drop manipulation. As soon as Leibniz has found a valid target, the application displays a box around the target portion of the expression and displays a message in the message panel indicating what manipulation the system thinks the user is trying to accomplish.



As soon as the user releases the mouse button, Leibniz sends the expression to the kernel along with a request to perform the desired manipulation. When the result comes back, Leibniz formats and displays the result. If multiple manipulations are possible, the most likely manipulation gets displayed in the message panel followed by the ellipsis character (...) to indicate that more options are possible. To access alternative options during a drag-and-drop operation, the user holds the mouse pointer stationary for a brief period to pop-up a menu of alternative options.

## ■ The Architecture of the Leibniz System

The behavior described above is implemented through a combination of functions performed by the application and the kernel. Roughly speaking, the application contains the code necessary to translate expressions to and from a form that the kernel can understand, while code in the **LZCompute** package implements all of the necessary pattern matching and calculation tasks.

The application is responsible for parsing mathematical expressions and building an internal tree structure to reflect the structure of the expression. In addition, the tree structure also contains information about the location of each subexpression. As the user drags the mouse, the application uses this information to map the current mouse location to the portion of the expression the mouse is in. When results come back from the kernel, the application translates them from *Mathematica* form to a form suitable for display.

Most of the interesting behavior in this system is implemented in the **LZCompute** package. This is due to a conscious design decision. There were many manipulations and calculations that could have been performed either in the application itself or by code contained in the package. Whenever the option came up to either hard code a capability in C++ in the application or as *Mathematica* code in the package, I chose to implement the operation in the package. This provides two obvious benefits: the code in the package is easier to modify, and end users with a knowledge of *Mathematica* can modify the behavior of the system to suit their needs.

## □ Implementing Drag-and-Drop Calculations

When the user clicks and drags inside an expression, Leibniz sends a request to the kernel to evaluate a potential drag-and-drop operation. It does this by calling the function **LZEvaluateShuffle** defined in the **LZCompute** package. For example, to evaluate the dragging operation shown in the illustration above, Leibniz would call the following.

```
LZEvaluateShuffle[LZMover[x] + y == LZTarget[2 x]]
```

Note the **LZMover** and **LZTarget** calls wrapped around parts of the expression. These function calls are used to indicate which part of the expression is being dragged (the mover) and which part of the expression is the target. **LZEvaluateShuffle** returns a list of code numbers corresponding to actions that are possible.

If **LZEvaluateShuffle** had to just return a single response code, life would be pretty simple. We could make a series of definitions like this.

```
LZEvaluateShuffle[
  Plus[LZMover[a_], b_] == LZTarget[c_] ] := 110;
```

To indicate how **LZEvaluateShuffle** should respond to a particular dragging request. In order to return a list of matches, we have to be a little more clever. This is how **LZEvaluateShuffle** is actually implemented.

```
LZEvaluateShuffle[expr_] :=
  FixedPoint[(EvalShuffle[expr, #]) &, {}];
ES[expr_, n_] := (EvalShuffle[expr, list_] :=
  Append[list, n] /; FreeQ[list, n]);
```

**LZEvaluateShuffle** is implemented as a fixed point calculation starting with an empty list and repeatedly applying the **EvalShuffle** function. Rather than return a single code number, **EvalShuffle** appends its code number to the list only if the list does not already contain that particular code number. **EvalShuffle** is defined via a series of calls to the **ES** function: each of those calls defines **EvalShuffle** for a particular pattern. For example, the **ES** rule that matches the drag request shown above is the following.

```
ES[Plus[LZMover[a_], b_] == LZTarget[c_] ] = 110;
```

How does Leibniz know which subexpression to mark as the target? After all, a given element in a mathematical expression is a member of multiple possible subexpressions. When the user moves the mouse to a new location in an expression, the application uses the information stored in the parse tree to map the location to a subexpression, typically a leaf in the parse tree. Leibniz then calls **LZEvaluateShuffle** with that subexpression identified as the target. If the result comes back empty, Leibniz moves up to the next largest subexpression containing the mouse location and calls **LZEvaluateShuffle** again with that subexpression identified as the target. This process continues until an action is found or there are no more possibilities for the target expression. Despite the fact that this process often requires multiple calls to **LZEvaluateShuffle** via *MathLink*, the process is fast enough to work comfortably in real time as the user drags the mouse over the expression.

Once Leibniz gets a list of code numbers back from the kernel, it calls the **LZMessage** function to retrieve descriptive strings for each of the potential actions. The descriptive string for the most likely option gets displayed immediately in the message panel. For example, the **LZMessage** entry that corresponds to the **ES** entry shown above is this.

```
LZMessage[110] = "Subtract from both sides"
```

As soon as the user releases the mouse button, Leibniz sends a request to evaluate the dragging operation. The **LZShuffle** function handles this request. The first

parameter to **LZShuffle** is the code number for the desired action, and the second parameter is the expression being manipulated with mover and target identified. For example, the **LZShuffle** definition that matches the **ES** rule shown above is this.

```
LZShuffle[110, Plus[LZMover[a_], b_] == LZTarget[c_] ] :=
Plus[b] == c - a;
```

## ■ Controlling Evaluation

One of the characteristics of *Mathematica* and other computer algebra systems that educators sometimes find frustrating is the fact that these systems generate results without showing intermediate steps. Leibniz addresses this problem by giving the user very fine control over just what part of an expression gets modified and how drastically the expression gets modified in a single step.

By selecting only a portion of an expression to work on, or by dragging and dropping within a larger expression, the user gets to focus the calculation. Leibniz supports this focusing behavior by sending just the relevant part of an expression to the kernel. When the result comes back, it gets swapped into place in the larger expression it came from, and that expression is then formatted and displayed.

Even with this focusing behavior, special care is needed to avoid unwanted evaluation. Consider our seemingly innocent example of a rule for moving a term from one side of an equation to the other.

```
LZShuffle[110, Plus[LZMover[a_], b_] == LZTarget[c_] ] :=
Plus[b] == c - a;
```

This seems straightforward enough, but what happens when, say, **b** is an integral? The user's intention will be to subtract that integral from both sides without evaluating it, but *Mathematica*'s default evaluation mechanisms will cause **b** to be evaluated, thus making the integral disappear after subtracting it from both sides. One obvious solution would be to give **LZShuffle** the attribute **HoldAll** and replace the result above as follows.

```
Hold[Plus[b]] == Hold[c] - Hold[b]
```

This solves the problem of the unwanted evaluation, but then causes the expression to fail to evaluate correctly when the user wants it to. For example, if **c** were 3 and **b** were 2, the user would expect to be able to subtract 2 from both sides of the original equation giving a new right-hand side of 1. However, with both **c** and **b** held as shown here, the user would see a new right-hand side of the unevaluated expression  $3 - 2$ .

The lack of control offered by the standard evaluation mechanisms and the **Hold** mechanism make it necessary to find a more refined strategy for controlling evaluation.

The first element of this strategy is to force every expression that the system works with to be a purely symbolic expression. This is done by replacing all conventional mathematical functions such as **Plus**, **Integrate**, **D**, and so forth with purely symbolic analogs called **LZPlus**, **LZIntegrate**, **LZD**, and so on. Thus, rather than rendering the expression  $a+b$  as **Plus**[a,b], the Leibniz parser passes the expression to the kernel as **LZPlus**[a,b].

With a little bit of care, we can do pattern matching with these new expressions just as easily as we could with more conventional expressions. For example, by giving the function **LZPlus** the attribute **OrderLess**, we can write pattern matching rules for **LZPlus** without regard to the ordering of terms involved.

Because none of these new functions come with evaluation rules, everything gets held by default. To release the implicit hold, **LZCompute** uses a couple of mechanisms.

The first mechanism releases the hold partially so that the usual evaluation rules can perform obvious algebraic simplifications.

```
LZPEval[a_] :=
  a /. {LZPlus → Plus, LZTimes → Times, LZDivide → Divide,
        (LZPower[x_, n_] /; FreeQ[x, LZMatrix]) → Power[x, n],
        LZRadical[x_, n_] → Power[x, 1/n]};
```

Thus, the **LZShuffle** rule shown above becomes the following.

```
LZShuffle[110, LZPlus[LZMover[a_], b_] == LZTarget[c_]] :=
  LZPlus[b] == LZPEval[c - b];
```

This solves both of the problems described earlier. If **c** is 3 and **b** is 2, the right-hand side will correctly evaluate to 1. If **b** is an integral, **LZPEval** will leave it unevaluated in the **LZIntegrate** form.

The second evaluation mechanism used is a complete evaluation mechanism, which converts every function in an expression from its LZ form to its conventional form. This mechanism is implemented via a function called **LZEval**.

```
LZEval[a_] := MapAll[LZRelease, a];
```

The **LZRelease** function maps each individual LZ function to its conventional equivalent. For example, here is part of the definition of **LZRelease**.

```
LZRelease[LZIntegrate[a_, {b_, c_, d_}]] :=
  Integrate[a, {b, c, d}];
```

This hold and release mechanism provides other benefits, as well. Because **LZRelease** has to map the held forms to their conventional equivalents, we can take advantage of the need to do that mapping to provide more sophisticated handling for certain special cases. The best example of this is matrix multiplication. Experienced *Mathematica* programmers know that if we try to compute the product of a two by two matrix and a column vector by naively writing

```
Times[{{a, b}, {c, d}}, {e, f}]
```

we get the incorrect result

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

If we remember to use **Dot** instead of **Times**, everything works out just fine. Leibniz solves this problem for the user by using **LZTimes** for everything that looks like a product and then relying on the release mechanism to sort out the details.

**LZRelease** uses some simple and straightforward pattern matching to ensure that **LZTimes** maps to the appropriate equivalent.

```
LZRelease[LZTimes[a_]] := LZResolveProduct[a];
LZResolveProduct[a_] = a;
LZResolveProduct[a_, b_] :=
  a.b /; And[MatrixQ[a], MatrixQ[b]];
LZResolveProduct[a_, b_] :=
  a b /; Not[And[MatrixQ[a], MatrixQ[b]]];
LZResolveProduct[a_, b_, c_] :=
  LZResolveProduct[LZResolveProduct[a, b], c];
```

## ■ The Future of Leibniz

Leibniz is an evolving system. The rule base is continually being refined as Leibniz users discover the need for new kinds of calculations. The open architecture of the Leibniz systems makes it relatively easy to add rules for new kinds of calculations.

Another recent addition to the system is a dialog facility. *Mathematica* code in the **LZCompute.m** package can now call a number of functions that bring up dialogs in the Leibniz front end. One function brings up a dialog that prompts the user to enter

an expression. Another function brings up a dialog that will present the user with a list of choices for them to select from. By allowing the code in **LZCompute.m** to interact directly with the user Leibniz can begin to move beyond the simple imperative style of “put that there” or “compute this”.

Over the long term, Leibniz will function as a test bed for ideas in interface design. By pointing out the limitations of the pattern-matching, drag-and-drop interface, Leibniz can point the way toward better and more powerful interface mechanisms for symbolic calculation.

### About the Author

Joe Gregg is an associate professor of mathematics and computer science at Lawrence University in Appleton, Wisconsin. He maintains a Leibniz web site at [www.leibnizsoftware.com](http://www.leibnizsoftware.com).