Meet the developer: Damien Katz on the Rnext formula language

Interview by
Michelle
Mahoney

Level: All
Works with: Domino Rnext
Updated: 07/02/2001

The bulk of the enhancements made to the formula language for Rnext are attributable to the work of Damien Katz, a software developer at Iris. Perhaps it is his experience as a Notes and Domino consultant prior to joining Iris in 1997 that best explains why he has been so successful at delivering the functionality that is most in demand from Notes developers; he approaches development from a user's perspective. Here's what he had to say about the formula language enhancements for Rnext.

We also invite you to read the companion article, **Enhancements to the formula language in Rnext**, where changes for Notes/Domino Rnext are explained in detail.

What did you have to do to incorporate the new looping and list manipulation functionality that is present in Rnext?
We had to rewrite the compute engine, which is the runtime interpreter of the formula language.

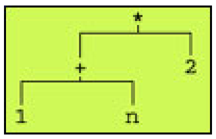
Rewrite it? Why was such a drastic change necessary?
The old engine, as it existed, was very simple. It was developed by Ray Ozzie years ago and was based on now outdated technology. We couldn't incorporate looping, nor many of the other enhancements users were requesting, until the runtime language was rebuilt from the ground up. The old architecture did not allow for it. The formula language compiled format is based on Reverse Polish Notation. This notation parses an expression using a set order of evaluation, and uses a stack to store and retrieve the values it computes. Here's how it works. If you have the following formula:

(1 + *n*) * 2

it is compiled into a machine readable Reverse Polish Notation format, similar to this:

1 *n*+ 2 *

The old engine begins to evaluate the expression left to right, so it places the 1 on the stack, then determines that the *n*—which let's say, is a field in the document—equals 3, and places the 3 on the stack. When it hits the plus sign, it goes back to the stack for two values and adds them, placing the result (4) on the stack. It then places the 2 on the stack and at the multiplication sign, retrieves the 4 and 2 from the stack and multiplies them. This process is somewhat inefficient and inflexible. The old engine had to do some interesting hacks to get @If to work. For Rnext, we have incorporated object-oriented programming into the engine. Now, the formula is instead transformed at runtime into an expression tree, as follows:



Due to the tree structure, the values are easier to store and retrieve and looping is possible. The tree still recognizes the order of evaluation; its furthest branches are executed first. In computer science, this is called a post-order traversal. In this example, the *n* is evaluated; it is added to 1, and the 4 and 2 are multiplied last.

What were your objectives in rewriting the formula language?
The primary goal was, and continues to be, to maximize performance gains. So far, we have done this in a number of ways.

First, there's caching results. This increases performance because you can cache the results of expressions that only need to be evaluated once. With the old architecture, no matter what the expression type was, you had to reevaluate it for each document in a view.

Next, we perform "lazy evaluations." This means that you can set aside a portion of a formula that does not need to be evaluated in a specific order and evaluate it only when its result is needed. Any @function that only operates on its direct arguments and does not rely on state information from anywhere else (@UpperCase, for example) can be lazily evaluated. @DbLookup and @Set cannot be lazily evaluated because they read from or write to the state of other, external variables. It is possible to differentiate between these two types of @functions and evaluate them differently because of the new tree structure of the formula language. All this is completely transparent to the end user, the formula programmer.

Also, treating internal data types as objects. The internal types in the language used to be concrete data types. Each @function knew the internal data structure of each item passed to it. So, for example, an @function would have to do pointer arithmetic to get at element *n* in a text list. Now that the internal types are object-oriented, the same @function does not need to use pointer arithmetic but instead, can ask the text list object for the value of its *n* th element. The new architecture allows for internals that have several different formats, making it easier to construct and manipulate the structures efficiently.

And finally, eliminating a lot of the silly limitations. For example, the new architecture rids you of the inability to assign a value to the same temporary variable twice without using @Set or to use the assignment operator inside another expression, like an @If expression. Together with the bigger things, like no looping and the 64k limit—all these limitations are now gone.

Are you seeing the fruits of all this labor?
Yes. In the context of the Web server and Notes client, the compute engine is now more than four times faster. In the context of rebuilding views, it is two to three times as fast. However, keep in mind that this does not mean that your views will refresh two to three times faster. Because disk I/O (reading notes from disk and writing results to the index) makes up a greater percentage of the time it takes to rebuild a view (maybe 80 percent depending on the size of the notes and the size of the results), the improvement to the formula computation portion has the effect of visibly improving the rebuild time by 10 to 15 percent. Also, the more formula intensive a form is, the bigger the performance gains. For example, a huge form with lots of hide-when formulas that might have taken a couple of seconds to refresh and been very sluggish, will now refresh almost instantaneously. This is assuming, of course, that the formulas don’t do lots of expensive things like @DbLookups and @GetDocField.

Which of the features that you've added is your favorite?
The array subscript operator, which allows the formula developer to get away from the dreaded @Subset. Consider what the developer used to have to do to get the fifth element out of a text list:

```
x := @Subset(@Subset( list; 5); -1);
```

Now it’s this simple:

```
x := list[5];
```

Plus it can be used after any expression, examples:

```
x := (list + 5)[x];
x := @DbName[2];
```

I think this will help a lot of developers be more productive, plus it might prevent a few cases of carpal tunnel syndrome.

About Damien Katz
Damien joined Iris in 1997. Before switching his focus to the formula language a year and half ago, he worked on developing several of the R5 templates, including the R5 Welcome Page (see his *Iris Today* article **Customizing the Welcome page**) and the Statistics & Events database template. Prior to joining Iris, Damien did Notes and Domino consulting in the Charlotte, NC area. When not working, he spends time with his wife Laura, plays basketball, lifts weights and runs his off-color humor Web site.

About Michelle Mahoney
Michelle Mahoney is a writer in the Notes/Domino User Assistance group and is currently working on updating and improving the formula language documentation for Rnext.