Simplifying your LotusScript with the Evaluate statement

by Mark Gordon

[Editor's note: This article resides in "Iris Today", the technical Webzine located on the http://www.notes.net Web site produced by Iris Associates, the developers of Domino and Notes.]

Would you like to write fewer lines of code? Let Domino do some of your work for you? I doubt you answered no, but if you have, what the heck -- read on to see what you'll be missing! We're going to show you how to use the LotusScript Evaluate statement to easily include short and powerful formula language constructs within your LotusScript code. You'll save a lot of headache, and as much as twenty lines of code at a time! A single Evaluate statement can often replace many lines of complicated script, and in some cases, can do what is nearly impossible in native LotusScript.

This article assumes you have some LotusScript experience, but does not assume you know a whole lot about the formula language. In fact, many developers who bring Visual Basic experience with them into the Notes world avoid the formula language. If you use LotusScript but know little about the formula language, you'll learn some good tricks in this article. If you developed Notes applications prior to Notes Release 4, when LotusScript was first introduced, you probably are already quite familiar with both the peculiarities and the power of the Notes formula language. But, you may not realize that you can leverage your formula language skills within LotusScript, building dynamic (yes, I said dynamic, even with Notes R4.5) formulas within your LotusScript routines.

This article will show you how the Evaluate statement works, and, just as importantly, give you examples of some of the more powerful formula language constructs you can exploit from your LotusScript routines. We will start with a basic Evaluate statement, then move on to focus mostly on dynamic Evaluate statements, which you may know were introduced officially in R4.6.1. However, we will also show you how to make Evaluate statements dynamic in a R4.5x or R4.6 environment as well (and look at why the Evaluate statement will not be necessary as often when using Notes R5). Finally, we will discuss several additional examples of the Evaluate statement, including one construct that actually gives you a faster-performing alternative to native LotusScript.

Downloading the sample database

You can try out the techniques described in this article by downloading the following self-extracting database (56Kb).

Evaluate.exe

A simple Evaluate statement

Let's start with the simplest possible example for an Evaluate statement: calling an @function that requires no variable or field arguments. The @Unique function, if used with no arguments, will return a unique character string such as this one: *MAGN-3WWMRX*. You can use @Unique to assign each document in a database an identifier that will always be unique, but which is much shorter, and thus, more manageable than the universal document ID. Many Notes developers store an identifier generated with @Unique in each Notes document. They're more manageable than the universal document ID, and they don't change when a document is cut and re-pasted, or when a replication or save conflict is resolved.

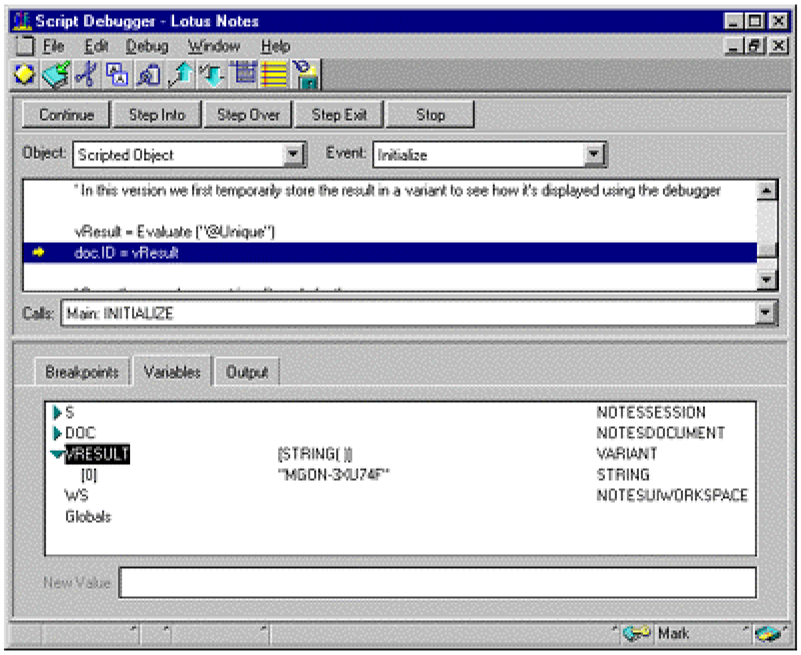
Within your applications, you can include on each form a hidden, computed-when-composed field called *ID*, with the simple formula @Unique. Each document gets a unique key when it is composed. However, in some cases, you may need to create a new document from an agent using a back-end LotusScript document object rather than from a direct user action. Unfortunately, LotusScript has no equivalent to @Unique. Here is an example of a script that creates a document and then uses an Evaluate statement to invoke @Unique and assign the unique ID. (**Note:** The code that declares and creates the new document is omitted here -- you can find the entire working agent in the *New Partner Document Via Script* agent in the sample database):

```
......

Set doc = s.CurrentDatabase.CreateDocument

......

' Use the Evaluate function to assign a unique ID using @Unique
doc.ID = Evaluate ("@Unique")
......
```

The above Evaluate statement simply invokes the @Unique function, which returns a unique identifier string. The string is then assigned to the *ID* field in the Notes document *doc*.

The results of an Evaluate are returned in an array

In the Evaluate statement above, we stored the result of the @Unique function in an item on the document simply by assigning the Evaluate statement to that item. If we were to instead assign the result to a variant and then examine the result using the debugger, we would find that the Evaluate results are returned in an array. Notice in the following screen that when we assign *doc.ID* to the variant *VResult*, the debugger shows that the result is stored as a single-value array (which is exactly how Notes items store single-value data in documents):



Object: Scripted Object
Event: Initialize

```
' In this version we first temporarily store the result in a variant to see how it's displayed using the debugger

vResult = Evaluate ("@Unique")
doc.ID = vResult
```

Calls: Main: INITIALIZE

```
S                                       NOTESSESSION
DOC                                     NOTESDOCUMENT
VRESULT          [STRING( )]            VARIANT
  [0]            "MGON-3XU74F"          STRING
WS                                      NOTESUIWORKSPACE
Globals
```

New Value

If the result of an Evaluate statement returns multiple values -- such as when doing an @DbColumn or @DbLookup (more on those later) -- the result is an array with the appropriate values.

String delimiters and dynamic formulas

The Evaluate statement executes everything within quotes as if it were a formula language formula. So, you can have multiple lines of code -- just don't include the carriage returns. In the above example, we simply put the formula in quotes. If you use formulas that have quotes within them, you can either use double quotes to delimit the entire formula and single quotes within the formula, or use an alternative string delimiter such as the { } brackets on either side of the formula and stick with normal double quotes within the formula.

Both of the following statements do the same thing -- they determine whether the string "Lotus" is a member of the list "Lotus," "IBM," and "Microsoft":

```
vResult = Evaluate (  "@IsMember ('Lotus' ; 'Lotus' : 'IBM' : 'Microsoft')" )

vResult = Evaluate (  {@IsMember ("Lotus" ; "Lotus" : "IBM" : "Microsoft")} )
```

Note: If you're one of those "avoid-@functions-like-the-plague" types, you may have forgotten that in @functions, each argument is separated by a semicolon, while elements hard-coded in a list are separated by colons. Of course, in practice, the list more often comes from a field or other variable than from a hard-coded set of values.

Up until Notes R4.6.1, the formula used in an Evaluate statement had to be determined **at compile time**. In other words, you couldn't use variable formulas. So, the following Evaluate statement (from the *Static versus dynamic formulas* form in the sample database) is valid in R4.6.1 and later releases of Notes, but not in a R4.5x client:

```
Dim strCompany As String
strCompany = "Lotus"
vResult = Evaluate (  {@IsMember ("} & strCompany & {" ; "Lotus" : "IBM" : "Microsoft")} )
Msgbox vResult(0)
```

You have to be careful with your quotes and brackets! Compare the first, static @IsMember formula with this second, dynamic one. In the dynamic version, we are constructing a string that the formula language interpreter sees as a static formula. The company names, as constants in that formula, have to be in quotes, but we are building that static formula using LotusScript variables. We have to start and stop the delimited formula with curly brackets to concatenate the variables into the right spots, but we still need to use the double quotes that the formula language requires around our variables, which it sees as constants. Whew! Kind of tricky to write and debug. Keep reading for a much easier approach!

The second approach to writing dynamic Evaluate statements works in most cases. You might find it less confusing than trying to keep your brackets and quotes straight. Also, while the official "dynamic" Evaluate statement does not work prior to R4.6.1, this approach does. This formula (also from the *Static versus dynamic formulas* form in the sample database) does the same thing as the previous one:

```
Dim s As New NotesSession
Dim doc As NotesDocument
Set doc = s.CurrentDatabase.CreateDocument
doc.tempCompany = "Lotus"
vResult = Evaluate (  {@IsMember (tempCompany ; "Lotus" : "IBM" : "Microsoft")}, doc )
Msgbox vResult(0)
```

The above formula is a lot easier to understand, because the formula looks like it would if you were writing it directly into a field formula or formula agent. In this example, we are telling the Evaluate statement to execute the formula we pass it *in the context of* a Notes document -- as if it were executing from a field formula on the form. The formula is static at compile time, but the properties of the Notes document it acts on are, of course, dynamic. That document can have items (fields) on it, and we can access those items from the formula. In this example, we created a temporary document (it's temporary only because we never saved it) just for the purpose of using variables in the @function.

In real examples, you can write values to the document object you are currently accessing in your script (you can remove those items after you use them if you don't want them saved in your document), or you can create a temporary document in your script just for this purpose. Most dynamic formulas will work using this approach, because in most formulas, having the field values and the arguments to the @functions dynamic is enough. However, if you want some other part of the formula to be dynamic -- if you wanted to dynamically construct the entire formula, for example -- only a R4.6.1 dynamic formula can work.

Where formulas beat script -- some more examples

Now that you know how to use the Evaluate statement, you may still be wondering why you would bother. Here are some more examples of things you can do with Evaluate statements.

Getting just the month or year from a date

When you convert a date to a string, the @Text function has a very powerful second argument that lets you specify how you want a date or time formatted in a string. The following is an example that automatically includes a year for a given date only if that date falls outside the current year. You'll find this example in the *Date Formatting With @Text* form in the sample database. Try creating a document with that form, and entering first tomorrow's date, and then your birth date. Tomorrow's date displays *without* the year, but your birth date displays *with* the year -- unless you were born yesterday!

```
   ' Let's say you're already working with a document and you want to format the date...
    Dim ws As New NotesUIWorkspace
    Dim doc As NotesDocument
    Set doc = ws.CurrentDocument.Document   'Shortcut alternative to explicitly creating a
NotesUIDocument

    ' A single Evaluate statement will format the date the way you like it
    ' @Text with a "D1" argument displays the year only if the date is a prior year
    vTheDate = Evaluate ({@Text (TheDate; "D1")}, doc)
    Msgbox vTheDate(0)
```

You can use the following codes for the @Text function's second argument (from the **Notes Help** topic on @Text):

- **D0 --** Year, month, and day
- **D1 --** Month and day, year if it is not the current year
- **D2 --** Month and day
- **D3 --** Year and month
- **T0 --** Hour, minute, and second
- **T1 --** Hour and minute
- **Z0 --** Always convert time to this zone
- **Z1 --** Display zone only when it is not this zone
- **Z2 --** Display zone always
- **S0 --** Date only
- **S1 --** Time only
- **S2 --** Date and time
- **S3 --** Date, time, Today, or Yesterday
- **S*x* --** Use when you cannot predict the exact format of the value being passed, but you know that it will be either a time, date, or both.

Appending items to a list

One of the big differences between LotusScript and the formula language is the representation of multi-value fields and variables. In LotusScript, a multi-value field is an array -- in fact, every field is treated as an array, even if it has only one value. In the formula language there are no arrays, only lists. The difference is that lists don't have to be declared or pre-sized, so they are much easier to work with.

To append a new value to a list of existing values, you can use a single line of LotusScript code -- an Evaluate statement -- which invokes the list processing features of the formula language and adds whatever appears in the *NewItem* field to the *List* field:

```
doc.List = Evaluate ( {List : NewItem}, doc)
```

Without an Evaluate statement, you would probably scratch your head for a few minutes and the come up with a block of code that looks something like this (see the form *Appending Items to a List* in the sample database):

```
Dim aList As Variant, intBound As Integer
' Set the current list into an array variable
aList = doc.List
' Redeclare the array to allow one more element
intBound = Ubound (aList)
Redim Preserve aList (intBound + 1)
' Set the new value into the new last element in the array
aList (intBound + 1) = doc.NewItem(0)
' Replace the document item with the new array
doc.List = aList
```

Another advantage of treating the fields as lists with the Evaluate statement is that you can do other "cleanup" work on the lists at the same time. This second Evaluate example uses @Unique to make sure that the new item it adds is not already in the list. It still uses a single line of code!

```
   doc.List = Evaluate ( {@Unique(List : NewItem)}, doc)
```

Now try doing *that* without the Evaluate statement!

Other uses for Evaluate
Here are some other examples of powerful @functions you can exploit in Evaluate statements:

- **@UserRoles --** Tells you what roles the user is a part of. This is useful for directing the processing based on a user's security level. The LotusScript equivalent is much more complicated, requiring you to use the NotesACL and NotesACLEntry classes -- and half a page of code -- to do what a single @IsMember ("[RoleName]"; @UserRoles) does.
- **@Replace --** Lets you write a single line of code to look through a list for one or more values you specify, and either removes them or replaces them with other values. Tricky syntax, but very powerful once you figure it out! There is no LotusScript equivalent in Notes R4.x.
- **@ReplaceSubstring --** Replaces portions of text from a string or list of strings -- again, this works with a single line of code, so there is no need to write a loop. There is no LotusScript equivalent in Notes R4.x.
- **@Left, @Right, @Middle --** Allow you to easily extract data to the left, to the right of, or between specified characters. LotusScript has a Left and a Right function, but they are not as flexible as @Left and @Right.

You'll evaluate less in Notes R5

Notes R5 contains enhancements to LotusScript that let you do several of the things we've discussed above without using Evaluate. Here are some of the new LotusScript functions mentioned in the **Notes R5 Beta 1 Release Notes**:

- StrLeft, StrRight, StrLeftBack, and StrRightBack do the equivalent of @Left, @Right, @LeftBack, and @RightBack.
- FullTrim is like @Trim
- Replace (a temporary name of the function) acts like @Replace
- ArrayGetIndex is like @Member
- ArrayAppend lets you append an item to a text list

According to the Release Notes, these are "LotusScript functions that correspond to some of the @functions. They are implemented to allow the script writer to manipulate Notes data more easily instead of achieving the same effect by calling Evaluate."

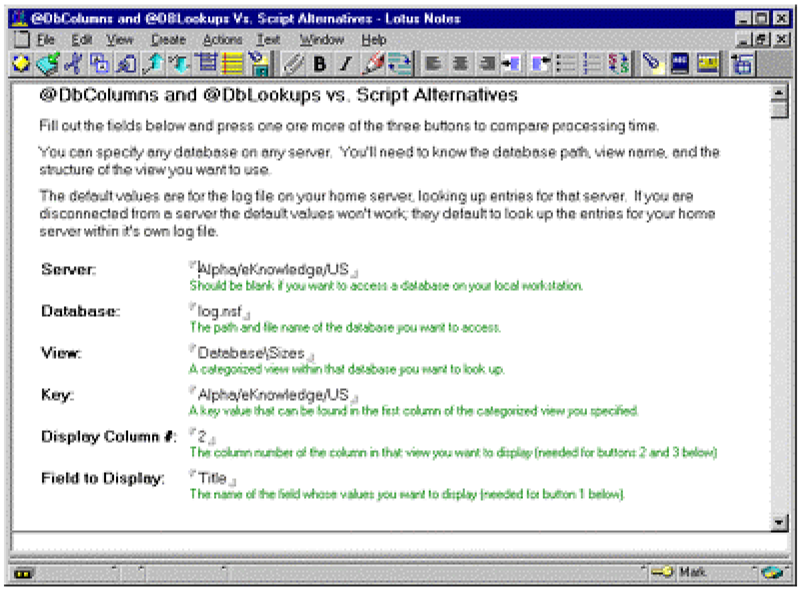
Sometimes the advantage is performance

Normally you wouldn't use an Evaluate statement to improve the performance of your application. A call to an Evaluate uses extra overhead to invoke the formula language compute engine in addition to the LotusScript interpreter. The performance degradation wouldn't normally be noticeable, but if an Evaluate call was executed while looping through thousands of documents, you might be able to measure a difference. However, there is one example of where using an Evaluate statement can actually make your code run faster -- a lot faster: using an *Evaluate ({@Dblookup...})* in place of the NotesDatabase method *GetAllDocumentsByKey*.

Let's say a script in a help desk application you're developing needs to access and store or display the name of each trouble ticket processed for a particular region. A common script developer's approach to accessing data such as this is to use the LotusScript method NotesDatabase.GetAllDocumentsByKey. This method would let you get all the trouble tickets from a view and then categorize them by region.

The reason the Evaluate is faster is because the @DbLookup gets values from a view index, without having to loop through and access each document. In contrast, pulling values using GetAllDocumentsByKey requires you to loop through and access each document.

In the sample database, you will find a form that illustrates how much faster using @DbLookup can be. The form is called *@DbColumns and @DbLookups Vs. Script Alternatives*. Here is what you will see if you create a document with that form:


@DbColumns and @DBLookups Vs. Script Alternatives - Lotus Notes
File Edit View Create Actions Text Window Help
Mark

@DbColumns and @DbLookups vs. Script Alternatives

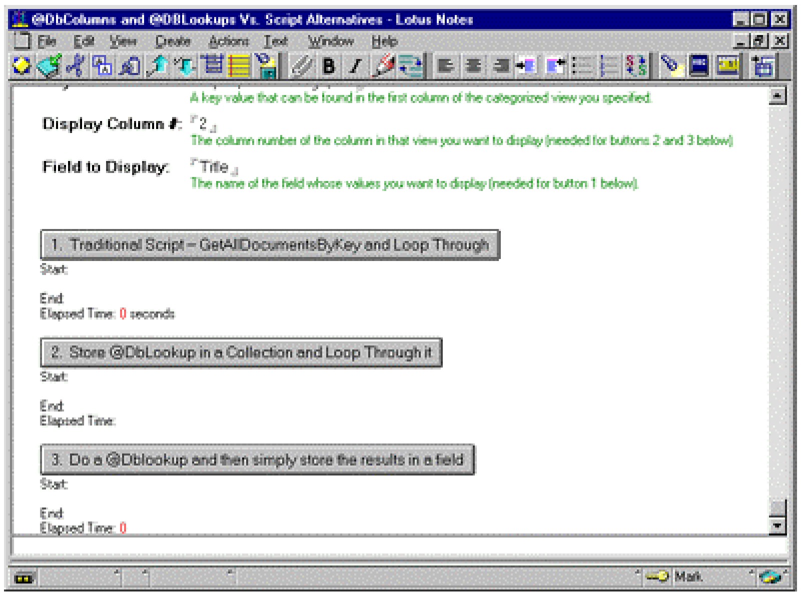
Fill out the fields below and press one ore more of the three buttons to compare processing time.

You can specify any database on any server. You'll need to know the database path, view name, and the structure of the view you want to use.

The default values are for the log file on your home server, looking up entries for that server. If you are disconnected from a server the default values won't work; they default to look up the entries for your home server within it's own log file.

Server:	Alpha/eKnowledge/US Should be blank if you want to access a database on your local workstation.
Database:	log.nsf The path and file name of the database you want to access.
View:	Database\Sizes A categorized view within that database you want to look up.
Key:	Alpha/eKnowledge/US A key value that can be found in the first column of the categorized view you specified.
Display Column #:	2 The column number of the column in that view you want to display (needed for buttons 2 and 3 below)
Field to Display:	Title The name of the field whose values you want to display (needed for button 1 below).

Once you fill in the information about the view you want to access, you can click one of three buttons: Traditional Script -- GetAllDocumentsByKey and Loop Through; Store @DbLookup in a Collection and Loop Through it; or Do an @DbLookup and then simply store the results in a field.


@DbColumns and @DBLookups Vs. Script Alternatives - Lotus Notes
File Edit View Create Actions Text Window Help
1. Traditional Script -- GetAllDocumentsByKey and Loop Through
2. Store @DbLookup in a Collection and Loop Through it
3. Do a @Dblookup and then simply store the results in a field
Mark

A key value that can be found in the first column of the categorized view you specified.

Display Column #:	2 The column number of the column in that view you want to display (needed for buttons 2 and 3 below)
Field to Display:	Title The name of the field whose values you want to display (needed for button 1 below).

Start:

End:
Elapsed Time: 0 seconds

Start:

End:
Elapsed Time:

Start:

End:
Elapsed Time: 0

The first button (Traditional Script -- GetAllDocumentsByKey and Loop Through) uses the GetAllDocumentsByKey method to access the documents matching the key you specify. Here is an excerpt from the button script:

```
...

Set view = db.GetView (thisdoc.TestView(0))

' Get a collection of documents in that view which match the specified key
Set dc = view.GetAllDocumentsByKey(thisDoc.Testkey(0), True)
...
```

The above script creates a collection of documents which match the key, and the following script loops through those documents one at a time:

```
 ' Loop through the collection, recording each returned value in the collection
     Set doc = dc.GetFirstDocument
     For x = 1 To dc.count
          ' Store the value of the field to be retrieved in the array
          vItemValue = doc.GetItemValue (thisDoc.TestSubjectField(0))
          aValues (x) = vItemValue(0)
          Set doc = dc.GetNextDocument (doc)
     Next x
```

When you create a document such as the one above and access a database on your network that has more than a few matches to the key you specify, you will see a *significantly faster* result with buttons #2 and #3, which use @DbLookup to retrieve the values. Here is an excerpt from button #2 (Store @DbLookup in a Collection and Loop Through it):

```
' Use a @DbLookup to get the results and store them in a variable
vList = Evaluate ( {@dblookup ("Notes" : "NoCache"; TestServer : TestDB; TestView; TestKey; TestDC)}, thisdoc)

' There is no need to loop through the results at all if we're just going to display them in a single field
' However, sometimes in real applications each return value needs to be processed.
' This loop will simulate that processing
Forall strName In vList
   x = strName
End Forall
```

When tested, button #2 ran in about one second, versus six seconds for button #1. That's quite a difference!

Of course, an @DbLookup has a 64K limit on the amount of data it can return. But in many cases, this approach is preferable because each lookup is relatively small. If you were writing a nested loop agent, for example, looping through hundreds or thousands of documents and executing a second nested loop to look for keyed matches for each document in the outer loop, it would really make a difference if each lookup ran six times faster! You could include an error handler to test for a given lookup failing due to the 64K limit, and branch to alternate GetAllDocumentsByKey code for the lookups that failed.

One more thing -- formulas "write" many of your loops for you!
Like the second button, the third button in the above example also uses an @DbLookup, but it adds some code to "clean up" the data. Instead of looping through the results in LotusScript, it uses a single Evaluate statement to implicitly "loop" through the data, removing any extra spaces from each item, and also removing the duplicates from the list. Here is an excerpt from the button script:

```
' Use an @DbLookup to get the results and store them in a temporary field
thisdoc.tempList = Evaluate ( {@dblookup ("Notes" : "NoCache"; TestServer : TestDB; TestView; TestKey; TestDC)}, thisdoc)

' Clean up the results (removing spaces and duplicate entries) and store the results
thisDoc.Results3 = Evaluate ( {@Trim (@Unique (tempList))}, thisdoc)
Call thisdoc.RemoveItem ("tempList")  ' no need to keep the temporary field
```

The @functions @Trim and @Unique, along with many other @functions, will implicitly loop through a list of items and operate on each element in that list. You may have written code to remove extra spaces from strings. @Trim does it just like that, and if you pass it a list, it does it on each item in the list. Likewise, many programmers have written code to extract unique items from a list. It's built-in with the formula language. By the way, in our benchmarks, button #3 ran twice as fast as button #2!

Notes R5 will help in this situation as well
The @DbLookup in an Evaluate statement runs faster, as discussed, because it lets you retrieve values from a view without accessing each document. There are enhancements to Notes R5 that will let you do the same thing without an Evaluate statement. The NotesView class has a new method, GetAllEntriesByKey, which has the following syntax:

GetAllEntriesByKey
***notesViewEntryCollection* = *notesView*.GetAllEntriesbyKey(** *keyArrayVariant* [**,** *exactMatchInteger*] **)**

NotesViewEntryCollection is a new class. You can navigate through a NotesViewEntryCollection much like you would a NotesDocumentCollection, except that you can avoid accessing each document and thus, improve performance. The advantage of the GetAllEntriesByKey method is that the resulting collection will return documents in the exact order that they appear in the view. Also, when accessing documents from a NotesViewEntryCollection, the ColumnValues property is available.

Conclusion

When Lotus included LotusScript in Notes R4, an object-based structured programming language became part of the Notes development environment for the first time. Many people thought LotusScript spelled the end of the formula language and its unusual @functions, but as most seasoned Notes developers know, the formula language complements LotusScript in many powerful ways.

Now you know how to get the best of both worlds, including powerful constructs from the formula language within the more structured environment of a LotusScript routine. We hope that these techniques will save you time and help you produce applications that are easier to maintain.