CHAPTER 10

Custom Function Primer

Custom functions are without a doubt one of the most powerful features in FileMaker; we cannot advocate their use strongly enough.

When added to a file by a developer, custom functions become available in various calculation dialogs as additional functions for use within expressions. They are snippets of code that, just as FileMaker's preestablished functions do, accept parameters and produce output. One example might be

```
fnCommission ( unitPrice; quantity; discount )
```

This function would presumably return a dollar amount based on some formula that multiplied unitPrice and quantity, subtracted a discount from the total, and then applied a percentage or some internal factoring to arrive at a sales commission.

Note that as a developer using a given custom function, you do not even need to know what that formula might be. All you require is that the function return a meaningful and consistent result when fed the necessary parameters.

Custom functions allow developers to abstract portions of code, independent from database schema or scripts, where it's then possible to reference a particular piece of logic throughout one's database. For example, if an organization's commission rates needed to change, the system's developer could edit a single custom function containing those rates, and all the calculations based on that function would immediately (depending on their storage settings) reflect and use the change.

Custom functions also facilitate code reuse: It's reasonably easy to copy code and insert a function into a different file. After being written and debugged, functions can be reused as necessary.

Custom functions can also serve as permanent "system variables" that are not subject to session issues as global fields and global variables are. The values within a custom function do not expire at the end of a user's session, they are consistent across all users of a database, and a developer can change them centrally as needed.

Note finally that FileMaker Pro 9 Advanced is required for authoring or editing custom functions. After a function is added to a file, however, it becomes available to any user or developer who has access to a calculation dialog, be it in the service of tasks such as defining fields, writing scripts, or even performing calculated replaces.

The Custom Function Interface

The Edit Custom Function dialog (see Figure 10.1) allows developers to define parameters that then serve as input for an expression written to reference those parameters.

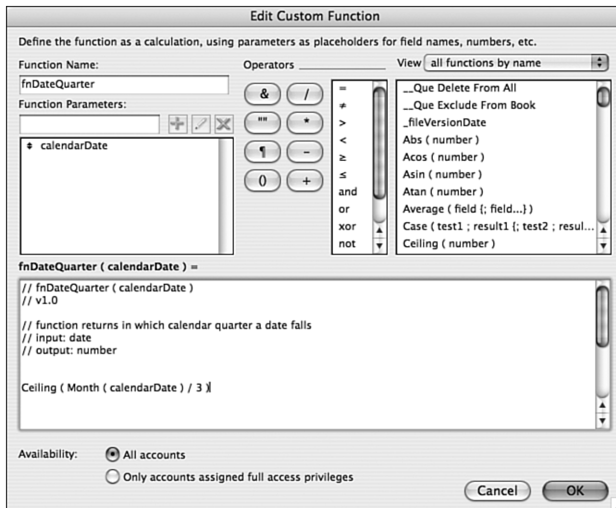

Figure 10.1

The Edit Custom Function dialog.

Custom Functions: Things to Remember

Custom functions work much like calculation functions, but it's important to understand the following aspects of custom functions:

- Custom functions follow the same rules for syntax that calculation functions follow.

➔ *For a review of calculation syntax,* ***see*** *Chapter 7, "Calculation Primer,"* ***p. 53****.*

- Instead of referencing schema information (data fields), custom functions use parameters. Values for these parameters are passed into the custom function from elsewhere in one's database via the Specify Calculation dialog.
- Custom functions return a single result.
- Custom functions cannot directly use or access container data.

- Custom functions can use all the functions built into FileMaker, including other custom functions and external functions.
- It is possible to make use of schema data fields by using the Evaluate() function. The following example illustrates a scenario where a sales commission is referenced:

```
fnSalesCommission ( unitPrice; quantity; discount )
// function to calculate the sales commission for various transaction totals.
// expected input:
//    unitPrice = dollar amount to two decimal places;
//    quantity = integer;
//    discount = any number (positive = discount)
// expected result: a dollar amount.

Let ([
   salePrice = unitPrice * quantity;
   total = salePrice - discount;
   discountPenalty = Case ( discount > 0;  .01; 0 );
   commissionPercent = Evaluate ( "ProductRate::Commission" ) - discountPenalty
   ]; // end variable declaration
   total * ( commissionPercent - discountPenalty )
)
```

- Just as custom functions can reference other functions, they can reference themselves as well. This allows you to write recursive functions in FileMaker. Keep in mind that recursive functions require an exit condition, or you end up with endless recursion and no result returned.

  The following is an example of a simple recursive function that reorders a carriage return-delimited list from bottom to top:

```
fnListBackwards ( valueList )
// function to reverse the order of a ¶-delimited list
// expected input:
//    valuelist = text values delimited by ¶
// expected result: a valuelist of text values delimited by ¶ in reverse order

Let ([
   numOfValues = ValueCount ( valuelist "");
   firstValue = LeftValues ( valuelist; 1 )
   remainingList = RightValues ( valuelist; numOfValues - 1 );
   resultList = Case ( numOfValues = 1; ""; fnListBackwards ( remainingList ) );
   ];
   resultList & firstValue
)
```

- If no exit condition in a recursive custom function exists or an error in logic occurs, the maximum number of recursions a function can make is 10,000. This assumes that it is a nested call that requires FileMaker to maintain a stack in memory of each recursion's result. FileMaker stops the recursive nest and returns a "?" as the result of any calculation using that function.

  If, on the other hand, you write a custom function so that the results of one recursion are passed into the subsequent recursive call as a parameter (and thus not requiring FileMaker to maintain a stack of results, but rather only calculating results in a series), the maximum limit is 50,000. This technique is referred to as *tail recursion*; here are some simple examples to demonstrate:

```
fnSummation ( number; iterations; startValue )
```

  This function adds *number* to itself for as many times as *iterations* dictates. For example, fnSummation (5; 3; 0) adds 5 + 5 + 5 to return 15. fnSummation (5; 3; 4) adds 5 + 5 + 5 beginning at 4 to return 19.

  Nested technique, requiring that FileMaker "stack" the results in memory:

```
Case ( iterations > 1;
            // iterate and then add the new number on the way down
    fnSummation ( number; iterations - 1 ; startValue ) + number ;
           // return the result
     startValue + number
)
```

  Tail Recursion technique where the results from one iteration are passed entirely into the next iteration, requiring no saved stack:

```
Case ( iterations > 1 ;
           // add the new number and then iterate
    fnSummation ( number; iterations - 1 ; startValue + number ) ;
          // return the result
    startValue + number
)
```

Recursive Techniques

In the first example, FileMaker has to preserve each nested result (in a stack) to derive its final result. In the second, the function passes in everything required through the parameters of the function and does not require evaluation of the entire stack. The first example will only work up to 10,000 iterations. The second can be evaluated in serial, up to 50,000 iterations.

To understand tail recursion, consider an analogy. Assume that you want to measure how many cups a body of water contains. One method would be to use a single 1-cup measuring cup, scoop out the water, and keep a count of how many times you

do so. This method, akin to stack recursion or nested recursion, requires that you keep track of how many times you've been through the process. Now imagine a second scenario where each time you measure out a new cup of water, you increase the size of your measuring cup. It starts as one cup, then holds two cups, and finally ends at the same number of cups the original body of water contained. You would not need to keep track of any count in the process: The result derived from your last iteration is the result you're looking for.

If you refer to the previous examples, notice that the first example combines a recursive call with an operator. In the third line of the function, *number* is being added to the recursive result of fnSummation again. This approach requires that FileMaker "keep track of" each result from each iteration of your recursive call to calculate the final result. It needs to create a stack of results and pass down the chain to the end point and back up again to calculate the final result. FileMaker literally processes each step of your recursion twice.

If you compare that to the second example, the recursive call in line three is not paired with an operator. It is simply called again with different parameters. It's a subtle but important difference. At the end of the recursive process, the results of the final recursive call are simply the result you're looking for. FileMaker can process each instance of the recursion loop and "forget" what the prior instance returned without processing the stack a second time.

The operator combined with your recursive call is a sign that you're not using tail recursion. Look to create recursive calls without operators, where you're simply passing new parameters into your recursive function, and your system will perform better and be capable of deeper processing limits.

- Custom functions can interact with global and local variables. In the case of local variables ($myVar), FileMaker stores values specific to a currently running script. In the case of a global variable ($$myVar), the value in the variable is updated and maintained throughout the file in a single user's individual session.

  If you want to create or set variables within a custom function, use the Let() function:

```
Let ([
   myInternalVariable = $$globalVariable + 1;
   $$newVariable = 1 + 1;
   result = $$newVariable + myInternalVariable
   ];
   Result
)
```

Custom functions are powerful tools for building abstract units of logic that can then be reused throughout a solution. After a custom function has been created (and tested!) it's easy enough to re-create it in other files for use throughout all your solutions. We strongly recommend you create a library of tools to refine and reuse over time. To that end, see Chapter 11, "Useful Custom Functions," for a selection of some of our favorites.