Creating Features out of Text

by John Mark Osborne
www.databasepros.com

I thought about titling this third article in the Philosophy of FileMaker series, "Parsing Text" but, it sounds boring. Manipulating text is so much more than cleaning up imported or improperly inputted data. Text manipulation allows a developer to create features that aren't possible with a single FileMaker tool. The real power of FileMaker exists in combining tools much like a carpenter uses multiple tools to build a house. The Philosophy of FileMaker teaches you how to use these tools to create your own unique solution. As you read this article, keep reminding yourself that you don't want to memorize the techniques presented but imbibe their essence. Even though this article is designed to teach rather than tell, it's important to remind yourself while you are reading that there is a difference between regurgitating techniques and truly understanding their abilities.

The Basics

Before diving into feature creation, it's important to discuss the foundation of text parsing. When I started writing this article, I decided to look up the word "parse". Several dictionaries later, I finally found one that gave a modern day definition appropriate to the computer sciences. The word "parse" is defined as, "to analyze or separate (input, for example) into more easily processed components." For the average database developer, parse means to separate first and last names into separate fields. And, this is where our journey begins. Before discovering how to create features with text parsing, foundational concepts need to be studied.

Let's say you just imported a whole bunch of contact information into a FileMaker Pro database. Unfortunately, all the first and last names are contained within a single field, making sorting by last name impossible. For the sake of a simple example, let's say all the names contain a single word for the first name and a single word for the last name. There are more sophisticated formulas for parsing multiple word last names, middle names, titles and more but, that's not the focus of this article.

The best method for studying a text parsing routine is to use example data. Let's use my name in a field called Name:

Name = John Osborne

The simplest way to parse my name into separate fields is to use the Left and Right functions:

Left(Name, 4)

Right(Name, 7)

The Left function grabs everything from the first character to the fourth character while the Right function grabs seven characters starting from the end. The result for the first formula is "John" and the second is "Osborne". However, the formulas are not flexible enough to handle first names that are less or greater than four characters or, last names that are less or greater than seven characters. Luckily, FileMaker, Inc. provides the Word functions in FileMaker Pro 3.0 and later. Therefore, a more dynamic solution is the following:

LeftWords(Name, 1)

RightWords(Name, 1)

Word functions like LeftWords, MiddleWords and RightWords use an algorithm to determine where one word ends and another begins, thus, providing a dynamic text parsing formula. The essence of the algorithm is that spaces are word separators. What most people don't know is how much more complicated the algorithm is. For instance, spaces aren't the only characters identified as word separators and it's your job to know which ones. I'm not saying you should memorize the word separator character set but, it is important to know how to test your data against the Word functions. Simply place your formula into a Define Fields calculation and start entering sample data. It's important to test as many variables as possible within reasonable expectations of the type of data that will be entered. For example, try entering the following:

204.182.23.89

Osborne, John

jmo@filemakerpros.com

It's interesting to note that punctuation by itself is not considered a word separator but, punctuation followed by a space is considered a word separator. In addition, many characters are considered word separators even if they aren't followed by a space, such as an at sign in an email address. However, what's really important to learn from this lesson is that the Word function algorithm may not react exactly as you expect so you'd better test it with a sampling of the expected data entry.

Once you've tested your data and determine the LeftWords function will perform as required, you'll want to transfer your formula from a Define Fields calculation to a scripted calculation, like a Set Field or Replace step. Why? Because this formula doesn't need to recalculate each time the Name field is updated. Once you have parsed the first and last name data into separate fields there is no need for the Name field. A scripted calculation is better since it parses only when initiated and allows users to modify the data in the first and last name fields. One of the biggest mistakes made by inexperienced developers is using all of their formulas in Define Fields. Always pick the right place to use your formula based on how the data will be used.

Before FileMaker Pro 3.0

Back in the old days, FileMaker programmers didn't have the convenience of the Word functions. In order to parse a first and last name, it was necessary to combine the Left and Right functions with the Position function to produce the same results. What a relief to have the Word functions! But, I soon learned that these familiar old formulas still serve a very important role. The Word functions are great if the algorithm matches your needs but, more complex parsing solutions often require a nested function incorporating the Position function. Let's introduce you to a better way of parsing text with the same name example:

Left(Name, Position(Name, " ", 1, 1) - 1)

Right(Name, Length(Name) - Position(Name, " ", 1, 1))

These two formulas produce the same results as the LeftWords and RightWords functions. So, why would you want to go through all the trouble of writing a more complicated formula. The simple answer is control. The Position function enables you to specify any search character or string you desire. It enables you to design the algorithm rather than leaving yourself at the mercy of the Word function.

Let's examine these two formulas more closely. Anytime you want to study a nested function, work from the inside out. Starting with the first example, isolate the Position function nested inside the Left function. The Position function returns a number value representing the character position of a search value in a string and has four parameters. The first parameter is the text you are searching and can be provided by a string in quotes, a field reference or another function. In the example provided, the Name field is the text being searched. The second parameter is the search string and can also be provided by a string, a field or a function. In this example, there is a space between quotes to locate the word separator between the first and last name. The third parameter is the character position to start looking and is generally a value of "1" so the Position functions starts at the beginning of the text. The last parameter is the occurrence and is almost always a "1" as well since the first occurrence of the search string is typically the target.

It's always helpful to substitute actual data for the field references when working with a nested calculation in order to better understand how it works. You can then distill the result down much like an algebra equation:

Left(Name, Position("John Osborne", " ", 1, 1) - 1)

Since the space is located at the fifth character, the Position function returns a five as indicated below:

Left(Name, 5 - 1)

Subtract one from five and now you have the same Left(Name, 4) formula discussed earlier except that the number of characters returned by the Position function will change depending on the length of the first name. Even more fantastic is the ability to control the search string so you can modify this basic formula to meet any text parsing job.

Let's not forget the slightly more complicated function for grabbing the last name. The Right function works differently than other text functions, in that, it starts from the right side of the text. Since the Position function starts from the left side, the only way to marry the two formulas is figure out the length of the last name. This is done by subtracting the beginning point and the ending point. Since the last name is the last word in the field, the Length function is able to provide the ending point. Now all you have to do is subtract the Length from the position of the space to get the number of characters in the last name.

Memorize these fundamental text parsing formulas as they serve as the foundation for almost all text parsing tasks. In general, the Philosophy of FileMaker doesn't recommend memorizing information. It's better to remember the essence of a technique and where to find an example rather than regurgitating the technique. However, straight memorization in this case is okay since these formulas truly are the building blocks of all parsing tasks.

New Request Made Easier

Now for the moment you've all been waiting. Let's create a feature using text parsing. There are so many examples it's difficult to choose just one. I made the selection for this article based on a common developer mistake. Many developers offer a single find layout for all of a users searching needs. The layout comprises every single field a user might need to search rather than multiple layouts customized for each search task. For example, through interviews with a group of users, a developer might discover two predominant search tasks. Rather than creating one giant find layout, create a find layout for each of these tasks. This makes the interface easier to understand and the users happier.

Along the same lines is trying to teach casual FileMaker users how to create new find requests. If users have difficulty distinguishing the difference between browse and find mode, what do you think will happen with new requests? Wouldn't it be better to provide a simple request interface and create the new requests for the user? In the following example, the new request interface has been distilled down to a single field so as to teach the concept without a lot of bells and whistles. Once you understand the essence of this technique, you will be able to apply it to more complicated find tasks.

Let's start with the specialized find layout. Instead of entering find mode to perform the find, the layout for creating new requests will be shown in browse mode. Rather than regular fields, global fields are utilized for find criteria entry. Again, this example has been simplified so only one global field will be used. To make data entry as simple as

possible, the global field is formatted as a checkbox with all the possible find requests that need to be created.

The real magic comes when creating the script. Let's take a look at the entire script in figure 1 and walk through it step by step.

```
Allow User Abort [Off]
Set Error Capture [On]
Set Field ["x.find", ""]
Go to Layout ["Form View 2"]
Pause/Resume Script []
Enter Find Mode []
Loop
   Set Field ["a.text", "Middle("¶" & x.find & "¶",
                        Position("¶" & x.find & "¶", "¶",
                        1, Status(CurrentRequestCount)),
                        Position("¶" & x.find & "¶", "¶",
                        1, Status(CurrentRequestCount) +
                        1) - Position("¶" & x.find & "¶",
                        "¶", 1,
                        Status(CurrentRequestCount)))"]
   Exit Loop If ["PatternCount(x.find, "¶") + 1 =
                 Status(CurrentRequestCount)"]
   New Record/Request
End Loop
Perform Find []
If ["Status(CurrentError) = 400"]
   Show Message ["No find criteria was entered."]
Else
   If ["Status(CurrentFoundCount) = 0"]
      Show Message ["No records were found."]
      Show All Records
   End If
End If
Go to Layout [original layout]
```

Figure 1
New Request Made Easier - this script parses a checkbox formatted field into new find requests so the user doesn't have to know FileMaker.

The first two steps are pretty standard for most scripts and shouldn't need to be explained. Starting with the third step, Set Field initializes the global field where new request data is entered. This is done before displaying the special find layout to prevent the user from seeing previous find criteria. The script pauses at this point so the user can enter their find criteria by checking the appropriate boxes. When the user continues the script, it enters find mode where the magic begins.

The key to this solution is the Set Field step at the beginning of the loop. It's purpose is to transfer the values in the global checkbox field to new requests in find mode. Here is the formula again with returns inserted between each parameter to make it easier to see where one parameter ends and another begins:

```
Middle(

"¶" & x.find & "¶",

Position("¶" & x.find & "¶", "¶", 1, Status(CurrentRequestCount)),

Position("¶" & x.find & "¶", "¶", 1, Status(CurrentRequestCount) + 1) - Position("¶" &
x.find & "¶", "¶", 1, Status(CurrentRequestCount))

)
```

The purpose of this complicated formula is to grab each value from the checkbox. When multiple values are selected from a checkbox or, other value list formatted field, they are stored as a return separated list. For instance, if you select red, green and blue from a checkbox field, the values are stored in the following manner:

```
red¶
green¶
blue
```

In order to decipher a complicated formula, work from the inside out. Let's look at each parameter of the Middle function. Once each piece makes sense, the entire formula will be clear. The Middle function is similar to the Left and Right functions except that it extracts text from the middle of a piece of text. It is comprised of three parameters. The first parameter is the text being parsed. In this example, a concatenation of text and a field are used. The reference to the x.find field is the global field formatted as a checkbox where the user inputs their new request choices. But, why are the returns added to the beginning and end of the x.find field? The reason becomes clear if you consider a different example such as a return separated list of numbers:

```
21¶
1¶
6
```

If you search for a "1" in this list, two occurrences will be located since "1" also exists at the end of the number "21". If you place returns at the beginning and end of the list, you can search for a more unique value:

```
¶
21¶
```

1¶
6¶

Now it is possible to search for "¶1¶" and locate a single occurrence of "1". While this scenario is less likely with text values, it is possible and should be considered. Get in the habit of plugging up this programming hole or it may come back to haunt you later.

The second parameter of the Middle function tells FileMaker at which character to start extracting text. Instead of providing a static value, the starting point is the result of a Position function. The parameters of the Position function are very similar to the foundational example provided earlier. The main differences are the search string and the occurrence parameter. It makes sense to search for a return character since they surround the values we want to extract. But, what's up with the Status(CurrentRequestCount) function? The Status(CurrentRequestCount) function returns a number that corresponds to the current number of requests in find mode. This value changes as the loop creates each request in order to update the calculation to grab the next value in the return separated list.

If you understand how the starting point is located, the formula in the third parameter is very similar. The third parameter of the Middle function asks for the number of characters you want to grab starting from the value in the second parameter. In order to get the number of characters or distance, subtract the end point from the starting point (the formula would be a lot easier to understand if the Middle function wanted the end point). The formula for the starting point is easy since it is exactly the same as the formula in the second parameter. The ending point formula isn't much different since you want to locate the following return in the list. To modify the Position formula to locate the next return, just add "1" to the Status(CurrentRequestCount). This increments the current request count by one and locates the return at the end of the value you are trying to extract.

In order to prevent an endless loop, it is necessary to offer a step for exiting. There are only seven script steps that can exit a loop: Halt Script, Exit Script, Quit Application, Close, Exit Loop If, Go to Record/Request/Page [Exit after Last, Next] and Go to Portal Row [Exit after Last, Next]. You might consider the Go to Record/Request/Page step but, new requests are being made to accommodate the number of values in the return separated list rather than looping through a set of existing requests. That leaves only one other viable option, Exit Loop If. The question is, what condition will exit the loop at the right time? All you have to do is count the number of return characters in the list and compare it to the number of requests. When the number of requests equals the number of returns, all the value in the list have been parsed into new requests.

Once the loop is complete and all the values have been transferred to new requests, the find can be initiated. The rest of the script is fairly standard scripting merely providing error checking in case no records are found or no find criteria is entered. If the error

checking passes, the user is brought back to the layout where they started to view their search results.

Conclusion

If you decide to implement this technique in your own solution, it is likely you will use a more complicated combination of values to create your new requests. While some find criteria may differ from request to request, as demonstrated in this article, others may stay the same on each request, as was the case in a solution I created for a LASIK eye surgeon. My client wanted to provide a search screen for locating all of the possible surgery types without having to teach other surgeons how to create new find requests. However, some data, like age range, needed to be the same on every find request. The solution is simple. Create the first find request before entering the loop and after entering the loop, duplicate the find request instead of creating a new one, and only modify the values that need to change.

There are many other features that can be created using parsing routines. It's really up to your imagination. Many other parsing techniques are free to download from my web site such as parsing a CGI generated email, extracting email addresses, removing the current record from a portal based on a self-join relationship, highlighting find criteria in the found set and much more.