Technical Bulletin

Recursive Functions

Product:	ARPEGGIO™ R&R Report Writer® for Windows®	**Host:**	N.A.
		NIC:	N.A.
		Interface:	N.A.
Version:	All	**Oper Sys:**	Microsoft® Windows® NT®

Summary

This technical bulletin explains a powerful technique called recursive functions. In computer terminology, recursion occurs when a function calls itself. You may want to think of recursion as a form of looping, which occurs when a program performs the same thing repeatedly while a given condition is true.

Recursion and Looping

You may recall situations where you needed to perform a loop in a calculated field. Or, you may have no idea what we mean by recursion and looping. We'll explain and demonstrate recursion by presenting three user-defined functions (UDFs): substring replace, substring count, and substring search.

User-defined functions are created by choosing User Function from the Calculations menu. See the User-Defined Functions section of Using Functions chapter in the R&R User's Guide if you need help with this command. Also, you need to be familiar with the $ operator and the following functions: AT, LEN, MAX, SUBSTR, STUFF, and TRIM.

Substring Replace

We will create a function that allows you to replace all occurrences of a substring with another substring.

For example, dBASE® allows you to insert semicolons within a character field at the point where you want a new line to begin when the field is printed. R&R Report Writer supports this feature, but some users want R&R Report Writer to ignore all semicolons in a given field. The expression StrReplace(COMMENTS,";"," ") can be used to replace unwanted semicolons with spaces in a field named COMMENTS.

Here are the definitions for the two parts of the substring replace function. As we explain below, you need to break recursive functions into two parts.

Declaration: _ssReplace(c_string,c_old,c_new)

Formula: IIF(old$string, _ssReplace(
STUFF(string, AT(old, string), LEN (old), new), old, new), string)

Declaration: StrReplace(c_string,c_old,c_new)

Formula: IIF(TRIM (old)$TRIM(string),
_ssReplace(TRIM(string) ,TRIM(old), TRIM(new)), string)

The StrReplace() function requires three inputs: the string in which you are making the replacements, which is denoted by the name string, the existing substring you want to replace, denoted by the name old, and the replacement substring, denoted by the name new. The formula is straight forward; here is what it says in English:

> If the unwanted substring is contained within the string, replace all occurrences of the unwanted substring with the replacement substring, otherwise, return the original string.

The _ssReplace() function is called by the StrReplace() function to make the substring replacements. It repeatedly calls itself (loops) as long as there's still a copy of the unwanted substring contained within the string. We take advantage of the built-in STUFF() function to make the substring replacement.

Since the STUFF() function replaces a single occurrence of the unwanted string, each repetition of _ssReplace() calls STUFF() to make the next replacement.

When there are no more copies of the unwanted substring to replace, _ssReplace() returns the modified string. Here's what the formula says in English:

> If the unwanted substring is contained within the string, replace the next occurrence of the unwanted substring, otherwise, return the string.

Let's step through a simple example to get acquainted with recursion. Suppose you create a calculated field expression StrReplace("aba","a","A"). Here's what happens when the expression is evaluated:

1. StrReplace() tests whether the substring "a" is contatenated within the string "aba". This test is performed with the expression TRIM(substring)$TRIM(string). Both the substring and the string need to be trimmed so that trailing blank spaces are not included, otherwise you might look for the substring "a " (the letter "a" followed by four spaces) within the string "aba" and not find it.
2. Since the substring is contained within the string, the IIF() function evaluates the expression _ssReplace(TRIM(string), TRIM(old), TRIM(new)). This expression calls the _ssReplace() function with the trimmed string and two trimmed substrings.

3. The _ssReplace() function first tests whether the old substring is contained within the string, to decide whether to continue looping or terminate. The first time _ssReplace() is called, the substring "a" is contained within the string "aba". Since the condition is met, _ssReplace() calls itself.
4. In the second call to _ssReplace(), the STUFF() function changes the first "a" within "aba" to "A". During the second iteration of _ssReplace() the test is once again performed to determine whether "a" is contained within the string, which is now "Aba". Since there's still another "a" within "Aba", _ssReplace() calls itself again.
5. In the third call to _ssReplace() the STUFF() function changes the remaining "a" within "Aba" to "A". During the third iteration of _ssReplace() the test is once again performed to determine whether "a" is contained within the string, which is now "AbA". Since it's not, _ssReplace() returns the completed string "AbA".
6. The third call to _ssReplace() terminates and returns the completed string to the second call to _ssReplace(), which returns the completed string to the first call to _ssReplace(), which returns the completed string to StrReplace(), which returns the completed string to the calculated field. The calculated field expression StrReplace("aba","a","A") evaluates to the value "AbA".

General Observations about Recursive Functions

From this description of how a typical recursive function works, we can make the following general observations about recursive functions. We will apply these rules in the next two examples, also.

1. Recursive functions are separated into two parts: a function you use in calculated field expressions (let's call this the outer function), and another function used only by the outer function to perform the recursive calculations (let's call this the internal function). In the substring replace example, the StrReplace() function is the outer function and the _ssReplace() function is the internal function.
2. The outer function handles such tasks is pre-processing the input values and error checking. In the substring count example, input values are pre-processed by the TRIM() function, and error checking is performed by the $ (contained within) operator.
3. The internal function, the one performing the recursion, almost always uses the IIF() function to test whether to continue or stop the recursion. The CASE() function can sometimes be used in place of IIF() in very complex formulas.

Substring Count

The purpose of this function is to count the number of times a substring appears within a string. For example, you may have a ten character field that contains responses to ten Yes/No

questions. The contents may look like "YYNNYNNYNY". This function can be used to count the number of Ys or Ns.

Here's the definition for the substring count function, which like the StrReplace() function, is divided into two parts for the reasons described above:

Declaration: _ssCount (c_ss, c_string)

Formula:
```
IIF(ss$string, 1+_ssCount (ss,
        SUBSTR(string, AT(ss,string)+LEN(ss))), 0)
```

Declaration: StrCnt(c_ss, c_string)

Formula:
```
IIF(TRIM(ss)$TRIM(string),
        _ssCount(TRIM(ss), TRIM(string)), 0)
```

The StrCnt() function requires two inputs: the substring you are counting, which is denoted by the name ss, and the string in which you are searching, which is denoted by the name string. Here's what the formula says in English:

> If the substring is contained within the string, return the number of occurrences of substring within string, otherwise, return 0.

The _ssCount() function is used by the StrCnt() function to do the actual counting, if the substring occurs within the string at least once. Otherwise there's no need to call _ssCount(), so StrCnt() returns 0. Here's what the _ssCount() formula says in English:

> If the substring is contained within the string, return 1 plus the number of occurrences of the substring within the portion of the string after this occurrence, otherwise, return 0.

The _ssCount() function calls itself repeatedly until there are no more occurrences of the substring within the string. Each call to _ssCount() passes an ever smaller portion of the original string, throwing away the part up through the occurrence of the substring that was just counted.

Let's step through the mechanics of this function to make sure you understand it. For example, suppose you create a calculated field expression StrCnt("a","aba"). Here's what happens when the expression is evaluated:

1. StrCnt() tests whether the substring "a" is contained within the string "aba". This test is performed with the expression TRIM(ss)$TRIM(string).
2. Since the substring is contained within the string, the IIF() function evaluates the expression _ssCount(TRIM(ss), TRIM(string)). This expression calls the _ssCount() function with the trimmed substring and string.
3. The _ssCount() function first tests whether the substring is contained within the string, to decide whether to continue looping or terminate. The first time _ssCount() is called, substring "a" is contained within the string "aba". Since the condition is met, the count is incremented by 1 and _ssCount() calls itself.

4. In the second call to _ssCount(), the input string "aba" is shortened to "ba" since the first occurrence of "a" was counted. This is done by the expression SUBSTR(string, AT(ss,string) + LEN(ss)).
5. During the second iteration of _ssCount(), the substring "a" is contained within the string "ba". The IIF() function adds 1 to the count and calls _ssCount() again with the input string "ba" shortened to "" since the first two occurrences of the substring "a" were counted.
6. During the third iteration of _ssCount(), the substring "a" is not contained within the string "", so the recursion stops. The IIF() function returns 0, which is added to the 1 from the second call to _ssCount(), which is added to the 1 from the first call to _ssCount(), giving a total of 2, which is returned to the StrCnt() function. The StrCnt() function returns the value 2, since there are two occurrences of "a" within "aba".

Substring Search

The AT() function provides the ability to search a string for a substring and return its position within the string. The AT() function is limited to searching for the first occurrence of the substring. We will create a function that can search for a specified occurrence such as the first, second, etc. In addition, this function can search for the last occurrence, next-to-last occurrence, etc.

For example, suppose you've used the LIBNAME() function to return the report library path name, which might be a string such as "C:\RSW\REPORTS.RP6". To extract the library name from the path name you can use the SUBSTR() function provided you know the position of the last occurrence of the substring "\". To determine this position, use the StrSrch() function as follows.

The StrSrch() function requires three inputs: the substring, the string, and the number of the occurrence you wish to locate. To specify occurrences relative to the end of the string use negative numbers, such as -1 to specify the last occurrence, -2 to specify the next-to-last occurrence, etc. In the path name example, the expression

```
StrSrch("\",LIBNAME(),-1)
```

returns the position of the last backslash in the path name.

You can extract the library name from the path name by using the StrSrch() function in the expression SUBSTR(LIBNAME(),StrSrch("\",LIBNAME(),-1) + 1). The resulting string is "REPORTS.RP6". And in case you're wondering, the ".RP6" can also be removed from the path name to return the string "REPORTS". Here's one expression you can use to remove the path name and the file name extension:

```
LEFT(
        SUBSTR(LIBNAME() , StrSrch("\", LIBNAME(), -1) +1),
        AT(".", SUBSTR( LIBNAME(),STRSEARCH("\", LIBNAME(), -1)+1) -1)
```

Now that you've seen how the StrSrch() function can be used, we'll show you how it works. Here's the function definition, again in two parts.

Declaration: _ssSearch(c_ss, c_string, n_num, n_ptr)

Formula:
```
IIF(ss$(SUBSTR(string,ptr)) AND num>=1,
        _ssSearch(ss, string, num-1,
        AT(ss, SUBSTR(string,ptr)) + LEN(ss)+(ptr-1)), ptr - LEN(ss))
```

Declaration: Strsearch(c_ss, c_string, n_num)

Formula:
```
IIF(TRIM(ss)$TRIM(string),
        _ssSearch(TRIM(ss), TRIM(string),
        IIF(num<0, StrCnt(ss,string) +num+1, num), 1), 0)
```

The StrSrch() function requires three inputs: the substring you are searching for, which is denoted by the name ss, the string in which you are searching, denoted by the name string, and the number of the occurrence of the substring you are searching for, denoted by the name num. Here's what the formula says in English:

> If the substring is contained within the string, return the starting position of the specified occurrence of the substring within the string, otherwise, return 0.

Like the StrCnt() function, StrSrch() pre-processes the substring and string using the TRIM() function. It also pre-processes the occurrence number to convert negative numbers to positive numbers, and initializes a pointer to the first character in the string. The pre-processing of the occurrence number and the initialization of the string pointer deserve a more detailed explanation.

Occurrence number - Since you can't literally search for the -1st occurrence of a substring within a string, you need to adjust this negative number to the actual occurrence number. If there are two occurrences, using -l to specify the last occurrence actually refers to the second occurrence. Likewise, using -2 to specify the next-to-last occurrence actually refers to the first occurrence. The expression StrCnt(ss,string) + num +1 converts negative numbers to positive numbers.

String pointer - The _ssSearch() function uses a slightly different technique for processing the string than the _ssCount() function described above. The _ssCount() function chops off the portion of the string it has already processed. In contrast, the _ssSearch() function keeps a pointer to the position within the string up to which it has processed. The positioning method is offered as an alternative because it's a bit more complex, but operates faster.

The pointer is initialized by StrSrch() to 1, to point to the first character within the string, and is denoted by the name ptr. Here's what the _ssSearch() function says in English:

> If the substring is contained within the part of the string starting at the pointer, and there are more occurrences of the substring to search for, decrease the number of remaining occurrences to search for, move the pointer beyond the current occurrence, and search for the next occurrence, otherwise, return the position of the substring within the string.

This algorithm is a bit more complex than the ones we've worked with so far, so let's step through an example using the expression StrSrch("\",LIBNAME(), -1) where LIBNAME() evaluates to the string "C:\RSW\REPORTS.RP6".

1. StrSrch() tests whether the substring "\" is contained within the string "C:\RSW\REPORTS.RP6" with the expression TRIM(ss)$TRIM(string).
2. Since the substring is contained within the string, the IIF() function evaluates the expression that calls _ssSearch() with the trimmed substring and string, and the adjusted occurrence number, which in this case is 2. The occurrence number -1 denotes the "last" occurrence of "\", which is adjusted to 2 since the StrCnt() function finds two occurrences of "\".
3. The _ssSearch() first tests whether the substring is contained within the string and whether there are more occurrences to search for, to decide whether to continue looping or to terminate. The first time _ssSearch() is called, the string "C:\RSW\REPORTS.RP6" does contain the substring "\" and there are still two occurrences of "\" to search for. Since the conditions for recursion are met, _ssSearch() calls itself.
4. In the second call to _ssSearch(), the occurrence number is decremented from 2 to 1 since an occurrence of the substring "\" has been found. Also, the pointer to the string "C:\RSW\REPORTS.RP6" is incremented from 1 to 4, so the remaining portion of the string to search is "RSW\REPORTS.RP6".

   During the second iteration of _ssSearch(), the substring "\" is contained within the string "RSW\REPORTS.RP6" and there is still one occurrence of "\"to search for. Since the conditions for recursion are met, _ssSearch() calls itself again.
5. In the third call to _ssSearch(), the occurrence number is decremented from 1 to 0 since another occurrence of the substring "\" has been found. Also, the pointer to the string "C:\RSW\REPORTS.RP6" is incremented from 4 to 7, so the remaining portion of the string to search is "REPORTS.RP6".

   During the third iteration of _ssSearch(), the substring "\" is not contained within the string "RSW\REPORTS.RP6", and even if it were, there are no more occurrences of "\" to search for. Since the conditions for recursion are **not** met, _ssSearch() decrements the pointer from 7 to 6 to point to the second occurrence of the substring "\" with the expression ptr - LEN(ss).
6. The recursion unwinds as the third call to _ssSearch() returns to the second call, which returns to the first call, which returns to the outer function, Strsearch(), which returns the

value 6. The second occurrence of the substring "\"within the string "C:\RSW\REPORTS.RP6" is located at character position 6.

Conclusions

The three substring functions presented in this article were designed to have a balance of clarity, speed, and usefulness. You may wish to tailor these formulas to your specific needs. For example, you may wish to sacrifice usefulness for speed in the substring search function if you only ever need to search for the last (rightmost) occurrence of a substring within a string.

Instead of using the StrSrch() function, you could develop a dedicated function that is less general but faster. The following Rat() function is analogous to the AT() function, but returns the position of the rightmost occurrence of the substring within the string.

Declaration: _rat(c_ss, c_string, n_ptr)

Formula:
```
IIF(ss$(SUBSTR (string, ptr)),
        _rat (ss, string,
        AT(ss, SUBSTR(string, ptr))+LEN(ss)+(ptr-1)), ptr-LEN(ss))
```

Declaration: Rat(c_ss, c_string)

Formula: MAX(_rat(TRIM(ss), TRIM(string), 1), 0)