Paper 87-25

Advanced SAS String Functions and a Little Recursion Protect Against Run-Time Errors

Amy Roehrig-Swinford, Trilogy Consulting, Waukegan, IL

ABSTRACT

A custom query tool written for a broad audience presents certain problems for the programmer. Some users are content with the point and click interface while power users want to type in code. In variable transformations such as logs and ratios, both approaches can result in errors if the data is at all dirty. We wanted to protect our system against mistakes in user-entered code as well as data problems, while still providing the flexibility of user defined variables. The new string functions rxparse and rxsubstr provide a clean approach to the problem that is easy to implement. This paper presents two SCL methods that utilize these functions to generate code-saving code: checks that prevent defined run-time errors.

THE PROBLEM

With a user base that stretches from analysts to marketers, our query tool has to meet a broad range of needs. One of the requirements for this system is the ability for users to define a variable on an ad-hoc basis. For a marketer this might be something as simple as % Paid Life to Date, or for a statistical modeler it could be an equation as long as this paper. Both of these situations present the same problem on a different scale. If % Paid is calculated as Total Ordered / Total Paid, then we get divide by zero errors when the customer is a freeloader who hasn't paid at all. Likewise, our statistician could be trying to take the square root of a negative number. Either way, our code would crash at run time, when all we needed to protect it were simple IF statements. Unfortunately, we can't depend upon our users to be sophisticated enough to include these necessary conditions in their code. Additionally, if the code is being generated by the application, we still need some way to provide these checks.

THE SOLUTION

Our solution was a method that checks for potential error conditions and inserts the appropriate conditional code. Based on what our users are doing, we need to check for division by zero, and the square root or log of negative numbers. This is not a simple matter of reading an equation from left to right, nested parentheses need to be taken into account. For example, suppose we have the following:

```
userVar1 = Var1 /  (1 – SQRT(Var2))
```

Reading left to right, our protective code would be:

```
If (1 - SQRT(Var2)) ^= 0  and
   (Var2) >= 0 then
```

But this would cause run-time errors if Var2 were negative. What we need is:

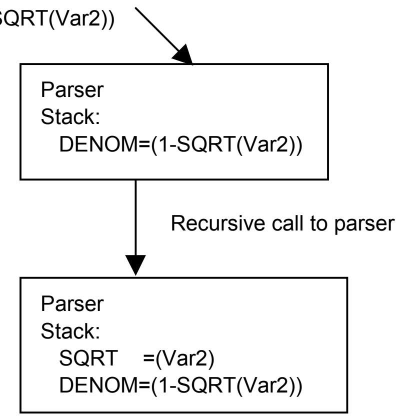
```
If (Var2) >= 0 and
   (1 – SQRT(Var2)) ^= 0 then
```

The reason that this works and the first does not is that SAS quits as soon as it fails a condition. So, we basically need to parse the code down to the innermost conditions and then work our way outwards. Anyone who has tried to do something like this with index and substr functions in a loop understands the true meaning of frustration. Luckily, there are new tools to make this job simple and straightforward.

RECURSION

The first issue to tackle is getting to the innermost condition. The simplest way to accomplish this is through recursion. For those unfamiliar with the term, recursive methods are methods that call themselves. Since our application was a graphical user interface, we merely added some SCL methods to our existing application. The method reads from left to right in the equation, looking for potential error conditions. When more are spotted, the parser calls itself with the remainder of the equation. This continues until all conditions have been handled. With each call to the parser, the necessary portion of the equation is added to a stack in the form of an SCL function. Using the equation in our example:

Var1 / (1 – SQRT(Var2))



While this is a simple example, the calls to the parser can continue to a deep level of nesting. Of course, this same functionality can be achieved through the use of loops, but the recursive calls result in far more concise code.

STRING FUNCTIONS

Now that we have a way to get to the inner tests necessary, we need a way to actually parse the equation. While this is certainly possible using only the old SAS® character functions such as index, substr and scan, the new pattern-matching functions rxparse and rxsubstr greatly simplify the task. The function of rxparse is to create a set of syntax rules for parsing strings. Once the syntax rules are assembled, different patterns can be pulled from the string with a single function call to rxsubstr. While the code for rxparse can get extremely complicated, it is well worth the effort. For example, in our case we needed to spot denominators, the SQRT function and the LOG function. We wrote our rxparse function to also recognize some other common equation components in the event we later add some checks for them. To meet our needs (plus a little), our call to rxparse is as follows:

```
rx=rxparse("`(@1 ‘ ‘* log $(30) ) #=4 |
   (@1 ‘ ‘* exp $(30) ) #=5  |
   (@1 ‘ ‘* sqrt $(30) ) #=6 |
   (@1 ‘ ‘* $n) #=1 | (@1 ‘ ‘* $d) #=2 |
   (@1 ‘ ‘* $f) #=3 | (@1 ‘ ‘* $(30) ) #=7 “);
```

This looks like gibberish, but it has meaning to the parser generator behind rxparse. What we have done here is to recognize the LOG function with up to 30 matched parentheses following it. When we see this pattern, we will assign a return value of 4. When we see the same pattern with the EXP function, we return a value of 5 and with SQRT a 6. We also recognize SAS® names by use of $n, integers (or digits) with $d, floating point numbers with $f and plain vanilla matched parentheses using $(30). By loading this into rxparse in this fashion, we can look for all of these patterns simultaneously. The variable on the left, RX can be considered to be the variable containing these syntax rules. Unfortunately, 6.12 can only hold one set of rules at a time, so we have packed everything we need into this one function, since a subsequent rxparse will eliminate the rule set from this one.

Once we have our rules, we can evaluate our equation for trouble spots. Since we are generating code from this, we are going to extract these patterns from the equation and load them onto the stack. To extract a pattern, the rxsubstr function has been provided:

```
   call rxsubstr(rx, subs, pos, len, detype);
```

The inputs to this function are rx (our rule set) and subs, which is our code to evaluate. The function returns pos, the start position of a pattern found; len, the length of the pattern found, and detype the number code for the type of pattern found. This way, we can grab a substring from our equation and be assured that the parentheses match and that it will be complete. There is no doubt in my mind that this is not only more succinct, but far less bug-prone than parsing would have been before the introduction of these pattern matching functions. In two lines of code we have spotted all of our potential trouble spots. With recursive calls to the rxsubstr code, we can wade through the entire equation quickly and easily. Once our stack is constructed, it is a simple matter to write the appropriate IF condition out to the code to be generated.

PUTTING IT TOGETHER

The resulting code ends up being more commentary than code. Essentially there are 2 methods. The first, GET_DENOMINATOR, assembles the syntax rules with the call to rxparse and reads the equation from left to right, calling the parser if needed. The second method is PARSE_DENOMINATOR, which recursively calls itself until the lowest level of nesting is achieved. And that’s it. PARSE_DENOMINATOR goes through a simple recursive loop, grabbing each substring identified by the rxsubstr method. The recursive call in PARSE_DENOMINATOR is shown here:

```
    *----------------------------------------
    Use new *experimental* rxsubstr function
    to quickly obtain denominators or
    function expressions. Rx contains output
    from rxparse, subs contains the string
    passed being examined. Pos is the start
    position of a recognized pattern, len is
    its length, and detype is the type of
    pattern we’ve found.
    ----------------------------------------;
    call rxsubstr(rx, subs, pos, len,
        detype);
    if (name='DENOM' and detype) or
       (detype=7) then do;
    *----------------------------------------
    Anything in the parse string is valid as
    a denominator. However, only matched
    parens are acceptable for the sqrt and
    log functions.
    ----------------------------------------;
        curdenom=substr(subs, pos, len);
        *-----------------------------------
        Add new condition to the list of
        conditions.
        -----------------------------------;
        denom_string=insertc(denom_string,
                curdenom, 1, name);
        *-----------------------------------
        If new condition also contains code
        which must be parsed for conditions,
        then recall parse_denominator
        recursively with just the new
        condition.
        -----------------------------------;
        if index(curdenom, '/') or
            index(upcase(curdenom), 'SQRT')
            or
            index(upcase(curdenom), 'LOG')
            then do;
          call send(_self_,
             'parse_denominator',  curdenom,
             denom_string, rx);
       end;

    *---------------------------------------
    Move on to the next part of the string
    beyond the denominator or expression.
    ---------------------------------------;
       if length(subs) >= len+1 then do;
          str=substr(subs, len+1);
       end;
       else do;
          str=' ';
       end;
    end;
    else do;
       str=subs;
    end;
```

This code uses rxsubstr to recognize any of the patterns defined in the rxparse function through the variable rx. The position and length of the recognized string are returned and used in a subsequent substr call to obtain just the relevant text. The type of pattern is returned in the variable detype. We determined what the values of detype would be in our call to rxparse when we used #n in the pattern definition. The string that meets the pattern requirements is added to the top of the SCL list denom_string. The name of the item will determine what type of check (e.g. ^=0, >0) will be done, with the text to be compared as the item in the list. If more potential trouble code is spotted using the index function, then parse_denominator, calls itself again. Each subsequent call to parse_denominator will add recognized strings to the top of the SCL list, creating a stack with the innermost conditions on the top. This list is then passed to another SCL function which generates the appropriate IF code using the name of the list item to determine what error condition to test.

WHY GO THROUGH THE AGONY OF FIGURING OUT RXPARSE? [CONCLUSIONS]

This is a difficult set of functions to use. Certainly I might have eventually been able to achieve my goals using the standard character functions. There are many reasons why I chose to use the more difficult pattern functions, but the main one is that it works! I am able to recognize the patterns I need and take action in comparatively few lines of code. Our application has a working parser that recognizes potential error conditions and inserts the protective IF statement, in the appropriate order. While there are some conditions we don't handle, such as taking a negative number to the ½ power (sqrt), we have the ability to easily add these patterns as they are identified. With a few short lines of code we have protected against the majority of errors that could have been caused by our users and data. And once again, our marketers don’t have to learn how to program to get what they want.

REFERENCES

SAS Institute Inc.(1997), *Pattern matching/changing expressions*, Cary, NC:SAS Institute Inc.

ACKNOWLEDGMENTS

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® Indicates registration.

Brand and product names are registered trademarks or trademarks of their respective companies.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Please contact the author at:

Amy Swinford
Trilogy Consulting
333 S 200 W
Lehi, UT 84043
Work Phone: (801) 766-9685
Fax: (801) 766-9686
Email: aimless@fiber.net