**Parsing a Compound Boolean Condition to Evaluate Its Clauses Individually and in Logical Groups**

*Daniel M. Himmel, Merck & Co., Inc.*

**Introduction**

The FDA statistician at last has received the long awaited CANDA (Computer Aided New Drug Application) on drug XYZ. With bated breath, he sits down at his computer console, eager to begin querying the clinical database which ABC Company has made available through the CANDA software. He has rehearsed this moment in his mind’s eye a million times, and without a moment’s hesitation he skillfully types in the first query. He wants a list of all male patients greater than 75 years of age who showed serious drug-related adverse effects. He presses ENTER and waits as the software begins to search the database. ABC Company had warned him that the database is huge, so he is not surprised when the query takes a over a minute to execute. At last, the answer flashes on the screen, and with an anticipat- ing smile he leans forward to read the results. But alas, his smile melts into an ashen frown, for the answer on the screen reads, "NO OBSERVATIONS FOUND". His mind begins to race. He must have made some mistake in his selection criteria -- but what? He realizes with a grimace that he will have to type in the query several more times, using slightly different selection criteria, until he is able to retrieve meaningful data. He changes the age requirement to 65 years and queries for males whose adverse experiences are 'possibly', 'probably', or 'definitely' drug related. After several minutes, he is relieved to find that the query has retrieved a list of patients for him, about 500 patients. He would like to get more information on the breakdown by age and on how many patients fell into each of the categories "possibly", "probably", or "definitely" drug related. Finally, he would like to see a separate breakdown for female patients. He realizes that to get this information will require him to run several separate queries and keep a running tally himself. "There has to be a better way," he thinks. "Wouldn’t it be nice, if I could run one query and get a concise table listing these breakdowns by selection criteria."

The program reviewed here was developed to address just such a scenario. The solution, as originally suggested by the FDA, was to keep a count of how many observations in the database satisfy each of the several selection criteria used in a query. The selection criteria were to be broken down into individual logical (boolean) conditions, and then the conditions were to be compounded step by step, with a count taken at each step, until the complete query was reconstructed. In this way, a list of conditions with corresponding counts would be presented when the query was complete. The challenge was to develop a self contained parsing-counting program that could be tested independently of the rest of the CANDA software and could be easily interfaced with it.

A careful algorithm, programming style, and set of data structures were devised which could operate effectively within SAS®. The program had to be a "black box" to the rest of the CANDA; that is, a database name and set of conditions could be packaged and sent to the program, which then would go about its processing and spit out an answer, without accessing any programs or variables from the rest of the CANDA software. In like fashion, the program itself was to be modular, consisting of separate macro routines which could perform well defined simple tasks, recursively if necessary. For input, arguments were passed to a macro routine. Since SAS macros do not return a value for an argument, such as is possible in PASCAL and other fourth generation languages, output from each macro routine had to be deposited in global macro variables (or sometimes in SAS data sets). Where possible, each macro used only local variables. However, several global macro variables were used to keep track of what part of the input compound boolean condition was currently being read, parsed, and processed. These special macro variables are listed in Table 1, and served such purposes as a position pointer, a counter for the current boolean operator being read, and an End Of Line flag.
Table 1: Important Global Macro Variables Used in Read-If.

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>PURPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;Clause</td>
<td>Boolean condition output from macros %ReadLeaf and %GetParen.</td>
</tr>
<tr>
<td>&amp;NewLeaf</td>
<td>Most recent boolean expression constructed and evaluated in %Check_op.</td>
</tr>
<tr>
<td>&amp;Position</td>
<td>Pointer for the current position in the input compound boolean condition string (&amp;A_string).</td>
</tr>
<tr>
<td>&amp;Op.obs</td>
<td>Number of the current operator/parenthesis to read.</td>
</tr>
<tr>
<td>&amp;Operator</td>
<td>Value of the current operator or parenthesis, e.g. AND, OR, ).,.</td>
</tr>
<tr>
<td>&amp;Oper_pos</td>
<td>&quot;Operator Position&quot;, the position of current &amp;operator in the compound boolean condition string (&amp;A_string).</td>
</tr>
<tr>
<td>&amp;EndParen</td>
<td>Logical flag to indicate when a close-parenthesis has been encountered for the current parenthesis pair.</td>
</tr>
<tr>
<td>&amp;EOL</td>
<td>&quot;End Of Line&quot;, logical flag indicating that the end of the compound boolean condition has been reached.</td>
</tr>
</tbody>
</table>

Data Structures and Bottom-up Refinement:

In the program, hereon referred to as Read-If, three chief data structures are used. The first serves as a vehicle for the actual compound condition. The entire compound condition is stored as one large character string in macro variable "&A.String" (Fig. 1). The second data structure is a list of boolean operators (AND, OR, parentheses), constructed to serve as delimiters for parsing out each individual simple condition. This operator list is stored as a character string in the macro variable, "&Op_list". Its structure (Fig. 2) consists of three spaces for each operator description, followed by three spaces for the numerical position of the operator in &A.String. The same operator information was originally stored in a SAS data set, but experience showed that storing it as a single macro variable substantially decreased execution time. The third data structure, used for output, is, at last, a SAS data set. This data set, called Leaf-List, contains the actual list of boolean conditions, as well as the number of observations satisfying each condition.

Bottom-up refinement is employed. Several simple macros are needed as rudimentary tools to be used by the main routines. For example, a macro called %ReadOp is required to read &Op_list. Since every six positions in &Op_list represents a new operator, the following equations can be used to read the current operator and its position in &A_string:

\[
\text{pos1} = [(\&Op\_obs - 1) \times 6] + 1 \quad \text{Equation 1}
\]
\[
\text{pos2} = \text{pos1} + 3 \quad \text{Equation 2}
\]

where &Op_obs is a positive integer recording which operator we are up to (e.g. 1, 2, ...), pos1 is the first position of the operator in &Op_list, and pos2 is the first position of the number for the location of that operator in &A_string.

Another example of a utility tool is %Calculat and the two macros it calls. %Calculat takes an input boolean condition (which could be simple or compound) and actually counts the number of observations in the input data set which satisfy that condition. The output is deposited in the data set LeafList. %Calculat calls the macros %BalanceP and %CutEmUp. %BalanceP balances the parentheses in the boolean condition if unbalanced. This prevents SAS syntax errors when using the boolean condition in a subsetting if statement. %CutEmUp substrings the condition if its string length is too long, so that the final output from the program can be presented in a legible tabular format.

For a summary of the macro routines, see Table 2 on the following page.
Table 2: Summary of Macro Routines Used in Read-If.

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Arguments Passed</th>
<th>Arguments Returned</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) %BalanceP</td>
<td>&amp;Condishn</td>
<td></td>
<td>Balances parentheses in current condition (&amp;Condishn).</td>
</tr>
<tr>
<td>2) %Calculate</td>
<td>Condishn</td>
<td>&amp;Clause</td>
<td>Counts number of observations satisfying current logical current logical condition (&amp;Condishn) and outputs results to the data set LeafList.</td>
</tr>
<tr>
<td>3) %Check_op</td>
<td>&amp;A_string, Leaf1</td>
<td>&amp;NewLeaf, &amp;EOL</td>
<td>Decides how to parse and evaluate &amp;A_string using logical precedence rules.</td>
</tr>
<tr>
<td>4) %Cut_EmUp</td>
<td></td>
<td></td>
<td>Substrings current condition string for output if string is too long to fit on one line. Deposits the substrings in an array which is appended to the data set LeafList.</td>
</tr>
<tr>
<td>5) %Drive_1</td>
<td>DataSet, &amp;A_string</td>
<td></td>
<td>Drives parsing and counting of &amp;A_string.</td>
</tr>
<tr>
<td>7) %MainDriv</td>
<td>DataSet, &amp;A_string</td>
<td>Screen/file output.</td>
<td>Front end and main driver.</td>
</tr>
<tr>
<td>8) %MakeList</td>
<td></td>
<td>&amp;Op.list</td>
<td>Constructs operators/parentheses lists.</td>
</tr>
<tr>
<td>9) %Print_It</td>
<td></td>
<td>Screen/file output.</td>
<td>Final output of the data set LeafList.</td>
</tr>
<tr>
<td>10) %Read_op</td>
<td>&amp;Operator, &amp;Oper_pos</td>
<td></td>
<td>Reads current boolean operator and its position from &amp;Op_list.</td>
</tr>
<tr>
<td>11) %ReadLeaf</td>
<td>&amp;A_string</td>
<td>&amp;Clause, &amp;Operator, &amp;Op.obs, &amp;Position, &amp;EOL.</td>
<td>Reads the current condition clause and the operator following it.</td>
</tr>
</tbody>
</table>

The Execution Flow:
The execution flow is directed by a macro routine called "%MainDriv", which has as arguments the name of the input data set, and &A_string (which contains the boolean condition string). %MainDriv performs some initializations. Then, it calls %Drive_1 to handle most of the processing, and finally it calls %Print_It to handle output.

%Drive_1 consists of two main steps. The first step is pre-processing, in which a list of all operators and their positions in &A_string is determined. This information is packaged in &Op_list by a small macro routine called, "%MakeList". In the second step, the compound condition in &A_string is read, parsed, and processed. These are accomplished by a recursive interplay of three macro routines, %ReadLeaf, %CheckOp, and %GetParen, which together constitute the workhorse of this program. %ReadLeaf reads the current simple condition clause and the operator following it. %CheckOp constructs the original compound condition from each of the simple conditions and decides which intermediate constructs need to be evaluated. %GetParen evaluates boolean expressions that are enclosed by parentheses.

The boolean condition stored in &A_string can be modelled by a binary tree graph [1], in which each boolean operator (i.e. AND, OR) is a node and each simple logical condition (e.g. A = 1, B > C, etc.) is a
Leaf (Fig. 3). In this configuration, a tree resolves into a compound boolean expression by in-order traversal [1]; that is, starting with the bottom leftmost leaf, you read left leaf, its parent node, and right leaf, and then move up to the next parent node and repeat this reading sequence (left branch, node, right branch), etc. In some languages, such as PASCAL [1] and C [2], the boolean expression can actually be stored in a binary tree structure. In SAS, however, this is not directly possible. We can still parse the boolean expression in the same order and retain some of the recursive functionality that a true binary tree would allow. We assume that &A String is a boolean expression of the form:

\[
(\text{Leaf}_1 \text{ Op}_1 \text{ Leaf}_2 \text{ Op}_2 ... \text{ Leaf}_{n-1} \text{ Op}_{n-1} \text{ Leaf}_n)
\]

where \( \text{Leaf}_1, \text{Leaf}_2, ..., \text{Leaf}_n \) are simple logical conditions (e.g. \( A=1, B>C, X\leq Y \), etc.) joined together by operators \( \text{Op}_1, \text{Op}_2, ..., \text{Op}_{n-1} \). Operators can have a value of either "OR" or "AND". Parentheses may be embedded in any of the "leaves" (\( \text{Leaf}_1, \text{Leaf}_2, ..., \text{Leaf}_n \)). However, for a simple case, let us first assume that there are no parentheses.

Our first task is to read and evaluate the first leaf. In this program, the leaf is read by \%ReadLeaf. \%ReadLeaf calls \%ReadOp to find out the position of the first operator, \( \text{Op}_1 \). \&A String is substringed from the starting position up to (but not including) \( \text{Op}_1 \). The position pointer is moved to the position right after \( \text{Op}_1 \), and \&Opobs (see Table 3) is incremented so that we are ready to read in \( \text{Op}_2 \) on our next call to \%ReadOp. Before leaving \%ReadLeaf, \%Calculate is called to evaluate \( \text{Leaf}_1 \). \%ReadLeaf returns, via global macro variables, \( \text{Leaf}_1 \) and \( \text{Op}_1 \).

Now, we want to look ahead to \( \text{Op}_2 \) so we can begin to compound our boolean expression and apply AND-OR precedence. This is the domain of \%CheckOp. \%CheckOp first calls \%ReadLeaf to get the next condition, \( \text{Leaf}_2 \), and the next operator, \( \text{Op}_2 \). Once we know the value of \( \text{Op}_2 \), we are guided through a decision tree (Fig. 4). If \( \text{Op}_2 \) is "AND", then we want to evaluate whatever follows it. We recursively call \%CheckOp again, which returns the boolean expression, \&NewLeaf. The beauty of this recursive approach is that we don't care whether \&NewLeaf contains one simple condition (i.e. \text{Leaf} 3) or a huge expression of leaves linked by AND's. It doesn't matter.

We take \&NewLeaf, whatever it contains, and build the following compound expression:

\[
\text{Leaf}_1 \text{ Op}_1 \&\text{NewLeaf}
\]

We then redefine \&NewLeaf to equal the expression in equation (4), and we pass \&NewLeaf to \%Calculate for evaluation. Thus, we have just constructed and evaluated the expression:

\[
\text{Leaf}_1 \text{ Op}_1 \text{ Leaf}_2 \text{ AND} ...
\]

Suppose that \( \text{Op}_2 \) was "OR" instead of "AND". In that case, we only want to evaluate expression (6) below:

\[
\text{Leaf}_1 \text{ Op}_1 \text{ Leaf}_2
\]

Thus, we let \&NewLeaf equal expression (6) and we pass it to \%Calculate for evaluation. Notice that it doesn't matter whether \( \text{Op}_1 \) is "AND" or "OR". An example of a run of Read-If without parentheses is shown in Fig. 5.

Now, let us assume that \&A String contains parentheses. The strategy used in the Read-If program is to have \%ReadLeaf call \%GetParen whenever the next operator turns out to be a parenthesis. \%GetParen evaluates everything inside the parentheses and returns in place of a leaf an entire expression bracketed by parentheses. We will need to employ two logical flags, \&parenth and \&endparen. \&parenth is a local macro variable which has a value of 1 whenever an open-parenthesis has been encountered. \&endparen is a global macro variable which has a value of 1 only when a close-parenthesis has been encountered. It will become clear shortly why we need both of these, one local and one global. There are four different cases in which we may encounter parentheses in a syntactically correct boolean expression. These are listed in Table 3.

Table 3: Types of constructs in which parentheses may be encountered. Three global pointers are shown with their positions immediately after evaluating the given construct. Notation: \( l = \) location of the beginning of the boolean expression string being evaluated; \( p = \) \&Oper.Pos, the pointer for the current operator position; \( p = \) position pointer.

| 1 | \( \text{Leaf} \) \text{ Op} ... | A close parenthesis follows an expression. | \( l \) op |
| 2 | \( \text{(Leaf) Op} ... \) | A single leaf enclosed in parentheses. | \( l \) op
| 3 | \( \text{Leaf Op ... paren} \) | An expression follows a single open paren. | \( lp \) o |
| 4 | \( \text{((Leaf ...) \text{ Op ... paren ... Op) ... paren ... Op ...) \text{ Op ... paren) ... paren} \) | Two or more open parentheses occur in a row. | \( lp \) |
In the simplest case, as in example 1 in Table 3, a close-parenthesis has been encountered. Thus, &parenth is false (= 0), &endparen is set to true, and the leaf immediately preceding the open-parenthesis is parsed using the global macro pointers and %SUBSTR, a SAS macro function. The leaf is output just as %ReadLeaf would.

In the next simplest case, exemplified by example 2 in Table 3, a simple logical condition without parenthesis immediately preceding the open-parenthesis is handled. The execution path is through Branch 2 and Branch 8. %Getparen is called recursively to handle the second open-parenthesis. It is worth noting that any number of adjacent open-parentheses can be handled in this way. For each additional open-parenthesis, we add one more %Getparen call to the recursion stack.

Testing:
To test the Read-If program, a front end was appended which created a hypothetical data set against which to query. This data set, "FakeBase", consisted of six numerical variables (A, B, C, D, X, Y) and two character variables (RELATION and SEX). FakeBase contained about 3000 observations. For &A_string, various different inputs were supplied to test each branch of the decision trees in %Check_op and %Getparen. Some examples are shown in figures 5 and 7.

Read-If was tested on an IBM 3090 mainframe under TSO in batch jobs. CPU time was recorded for each type of input. Execution generally required about five CPU seconds on average, which translated to about a minute in real time. Obviously, execution time increased with each additional leaf or parenthesis pair added.

Conclusions:
Most difficulties in the coding were associated with resolving macro variables containing quotes or parentheses. The solution consisted mainly in the copious use of the %BQUOTE SAS macro function. Also, in &Op_list, 'OPN' was substituted for '(', and 'CPN' was substituted for ')'.

To improve time efficiency, SAS data steps were used as little as possible. Excessive use of data steps would have slowed execution for several reasons. First, each new data step would have returned execution control to the SAS Supervisor [3], which would consume significant additional time. Second, recursive macro calls could not be made from inside a data step. Thus, extra flags would have been required to make sure that the correct macro routines were called at the conclusion of a data step. There were other disadvantages with data steps. As mentioned above, resolving macro variables that stored quotes or parentheses and the conversion of the same from data set variables back to macro variables presented difficulties. Also, it was generally found that macro routines without data steps required simpler algorithms.
This paper presented one way of parsing and evaluating an expression from inside SAS. This is obviously not the only way it could be done. For instance, with the introduction of SAS Release 6, it will soon be possible to accomplish the same task using code written in C. However, the Read-If methodology demonstrates the feasibility of parsing and evaluating on-line. It is hoped that this work may serve as a prototype for future endeavors of this sort.

**Acknowledgements:**
I would like to thank Merck & Co. and, in particular, James Bongiovanni, Nancy Bauer, and Joanne Trauth for their encouragement and support during the development of the work presented here.

**Footnotes**


(SEX EQ "M" AND AGE GE 40 OR SEX EQ "F" AND AGE GE 55) AND
(RELATION EQ "PROBABLY" OR RELATION EQ "DEFINITELY")

FIGURE 1: Example of a compound boolean condition such as might be stored in the macro variable &A_string. The compound condition is used as a selection criterion for subsetting data from the database.

Figure 2: &Op_list data structure. This macro variable records the positions of all operators and parentheses in &A_string. These are used as delimiters for reading each “Leaf” in the compound boolean expression.

Boolean Expression: A AND B OR C OR D AND E

FIGURE 3: Boolean expression represented by a binary tree graph, where A, B, C, D, and E are simple logical conditions (e.g. x>1, y< x, etc.). A, B, C, D, and E are called leaves, and the boolean operators occupy nodes. The graph can be resolved back into the boolean expression by in-order traversal of the binary tree.
FIGURE 4: %Check_Op execution. Displayed in pseudo-code is a flow chart of the decision branching which %Check_op uses to apply AND-OR precedence to decide what to evaluate. Macro routines are depicted in solid black boxes.

Figure 5: Example of output from Read-H program. The front end created a test data set called TEST_SET, which contained 2916 observations. Boolean conditions are listed on the left. The last condition listed comprises the entire compound boolean expression which served as input for the Read-H program. Execution, including generation of the data set TEST_SET, required 5.29 CPU seconds under TSO on an IBM 3090.
Figure 6: Decision branching in %GetParen. Shown is the execution flow in pseudo-code.

%GetParen parses and evaluates any expression inside parentheses by recursively calling itself, %ReadLeaf, and/or %Check_op. The input argument is &LocalPos, which is the first position of the expression to be evaluated. The output, like with %ReadLeaf, is &Position (the current position pointer), &Operator, and &Clause (the expression just evaluated).

**Figure 7:** Example of output from Read-If program using parentheses. The front end created a test data set called TEST_SET, which contained 2916 observations. Boolean conditions are listed on the left. The last condition listed comprises the entire compound boolean expression which served as input for the Read-If program. Execution, including generation of the data set TEST_SET, required 5.17 CPU seconds under TSO on an IBM 3090.