Expression Handling Functions: Part I
Unraveling the Expr(), NameExpr(), Eval(), ... Conundrum

Joseph Morgan, JMP Division of SAS Institute

Many beginning and intermediate JMP Scripting Language (JSL) programmers are unaware of the power of abstraction available from JSL expressions. Such meta-programming constructs are not always available in widely used programming languages such as C++ but are commonly found in functional programming languages such as Lisp. As it turns out, such constructs are particularly useful when the application being developed is complex. They facilitate process abstraction. Robert Sebesta (1999) describes abstraction:

> "The ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored. The degree of abstraction allowed and the naturalness of its expression is important."

This article attempts to unravel the mystery surrounding JSL expression handling functions and show how such functions can be used to solve nontrivial JSL programming challenges.

JSL Expressions

What exactly is a JSL expression? Chapter 3 of the JMP Scripting Guide (JSG) defines JSL expressions thus:

> "A JSL expression is any combination of variables, constants, and functions linked by operators that can be evaluated."

The key phrase here is "... *that can be evaluated.*" This means that each of the following is a JSL expression.

```
100.1                  //numeric literal
"string literal"       //string literal
x                      //variable (or name)
x & (y | z)            //logical expression
z*2 + z^2 -10 + pi()  //arithmetic expression
```

However, more complex examples like the following are also JSL expressions.

```
x = [];
for(i=1, i<=5, i++,
   x ||= random uniform(); show(x)
)
```

Although the term *script* is often used to refer to an example like this, it is really just an expression. Remember that the semicolon ";" is the *glue* operator that returns the value of its right-most argument. A script is nothing more than a single `glue()` function call with expressions as its arguments. To see this, notice that the previous example is equivalent to the following `glue()` function call.

```
glue(assign(x, []),
   for(assign(i, 1),
      less or equal(i, 5),
      post increment(i),
      glue(concat to( x,
                     random uniform()
                     ),
         show(x)
      )
   )
)
```

Hence, a JSL expression may be as simple as a literal or variable, but could be as complex as a script.

What is an Expression Handling Function?

A useful way to think of expression handling functions is as the set of JSL functions that enables you to regard expressions as data.

Functions such as `Expr()`, `NameExpr()`, `Eval()`, `Function()`, and `Recurse()` allow you to assign expressions to variables for later retrieval and possible evaluation. There are also functions that allow expressions to be assembled, disassembled, and probed. `Insert()` and `Remove()` are two of several functions that may be used to assemble and disassemble expressions whereas `Arg()` and `Head()` are intended for probing. JMP offers a full complement of these functions thus ensuring that JSL programmers can easily realize the abstraction by Sebesta (1999).

These functions (see *Table 1*) fall into two categories: those that evaluate their arguments when invoked and those that do not. The best way to understand this difference is to experiment with these functions. To follow along, launch JMP and run the code fragments presented in the following sections.

Table 1 JSL Expression-Handling Functions

Evaluate Arguments	Do Not Evaluate Arguments
`Parse()`	`Expr()`
`Eval()`	`NameExpr()`
`EvalList()`	`EvalExpr()`
`Function()`	`Arg()`
`Recurse()`	`NArg()`
`Substitute()/SubstituteInto()`	`Head()`
`Remove()/RemoveFrom()`	`HeadName()`
`Insert()/InsertInto()`	

Expression Handling by Example

The following questions were real problems presented by JSL programmers who had a task they were trying to complete. These challenges are not intended to represent the range of questions a typical JSL programmer is likely to face, but they comprise a series of typical and commonly encountered questions.

1. The Substitute() vs. SubstituteInto() Question

Suppose you want to write a script that invokes the distribution platform but the column to be analyzed is stored in a variable. In cases like this, the `Substitute()` or `SubstituteInto()` function may be used but it is sometimes not clear which one should be used.

For example, the following script uses `Substitute()` to replace `colx`, with `weight`, but fails.

```
//script 1
stmt = Expr(distribution(column(colx)));
x = "weight";
Result = Substitute(stmt, Expr(colx), x);
show(stmt); show(Result);
```

If you execute this script, the log shows:

```
Not Found in access or evaluation of 'distribution' ,
Bad Argument( {colx} ), distribution( Column( colx ) )
```

Because `Substitute()` evaluates its arguments, it attempts to evaluate `stmt`, but fails because `colx` does not exist. One solution is to properly *quote* the first argument of `Substitute()`. That is, use `NameExpr()` to retrieve the value of `stmt`.

```
//script 1 - revised
stmt = Expr(distribution(column(colx)));
x = "weight";
Result = Substitute(NameExpr(stmt),Expr(colx),x);
show(stmt); show(Result);
```

Now, execute this revised script to see the value of `stmt` and `Result` displayed in the log.

```
stmt:distribution(Column(colx))
Result:distribution(Column("weight"))
```

Alternatively, `SubstituteInto()` may be used. The difference is that, unlike `Substitute()`, `SubstituteInto()` does not evaluate its first argument but simply updates it in place.

```
//script 2
stmt = Expr(distribution(column(colx)));
x = "weight";
SubstituteInto(stmt,Expr(colx), x);
show(stmt);
```

When you execute this script, the result in the log is

```
stmt:distribution(Column("weight"))
```

Summary Points

Point 1:

A common JSL mistake is to assume that executing `Expr(x)` is equivalent to executing `NameExpr(x)`. Indeed, in the following example, these two statements return the same thing.

```
Expr(4 + 35)
NameExpr(4 + 35);
```

If you execute them one at a time, the log shows,

```
Expr(4 + 35);
4 + 35
NameExpr(4 + 35);
4 + 35
```

The result is the same for both statements. `Expr(x)` returns *its argument unevaluated* and `NameExpr(x)` returns *the value of its argument unevaluated*. The argument to `NameExpr(x)` should be a variable, but when it is an expression it simply returns its argument.

Consider the next statement.

```
x = Expr(2 + 50);
```

When you execute this statement the expression `2 + 50` will be stored in x.

Now consider the following statements.

```
Expr(x);
NameExpr(x);
```

Execute each statement and look at the log.

```
Expr(x);
x
NameExpr(x);
2 + 50
```

Since `Expr()` returns its *argument unevaluated*, the name `x` is returned, whereas `NameExpr(x)` returns the *value of its argument unevaluated* — `2 + 50`.

***Point*: Executing `Expr(x)` is not equivalent to executing `NameExpr(x)`.**

2. Obtaining Distinct Items From a List

Suppose you have a sorted list and want to retrieve only distinct items. There is no JSL function to accomplish this, but it is easy to script a solution.

Consider the following two lists.

```
Things = {"apple", "apple", "apple", "cat", "cat", "cat",
   "golden", "grape", "mango", "mango", "silver",
   "silver"};
Numbers = {1,200,200,200,400,400};
```

One approach is to iterate over items in each list and pick out the distinct items as the iteration progresses. However, here is an alternative and compact solution that illustrates the `EvalList()` function.

```
indx = {};
indx[1 :: NItems(Things )] =
   Expr( 0 == i++ | Things [i - 1] != Things [i] );
   i = 0;
distinctlst = Things [Loc( EvalList( indx ), 1 )];
```

Note that the second statement creates a list of logical expressions and that this list contains the same number of items as the sorted list. Each expression is intended to compare the corresponding entry in the sorted list to the item at its left. When evaluated (by `EvalList()` in the fourth statement), each expression in the list evaluates to either true or false. The `Loc()` function in the fourth statement converts this list of 0s and 1s into a vector of indices that retrieves the distinct items.

The following function is a more robust solution.

```
distinct list = Function( {lst},
  Local( {indx = {}, i = 0},
    If( Is List( lst ),
      If( N Items( lst ) < 2,
            lst,
            indx[1 :: NItems( lst )] =
            Expr( 0 == i++ | lst[i - 1] != lst[i] );
            lst[Loc( EvalList( indx ), 1 )];
        )
      )
    )
  );
```

Calling the function with the list as its argument gives the following unique items.

```
Distinct List(Things);
{"apple", "cat", "golden", "grape", "mango", "silver"}
Distinct List(Numbers);
{1, 200, 400}
```

Summary Points

Point 2:

When using the `Eval()` function, a common mistake is to assume that executing `Eval( x )` is equivalent to executing `x`. This mistake can be easily made if you examine examples like the one below, where the second and third statements produce the same results.

```
x = Expr(4 + 25);
x;
Eval( x );
```

The first statement stores the expression **`4 + 25`** in `x`. If you execute the second and third statements in turn, you see the following in the log.

```
x;
29
Eval( x );
29
```

However, what if the first statement was a nested `Expr()` function as in the example below.

```
x = Expr(Expr(4 + 25));
x;
Eval( x );
```

Note that, for this example, the first statement stores the expression `Expr(4 + 25)` in x. If you execute the second and third statements in turn, you see the following in the log.

```
x;
4 + 25
Eval ( x );
29
```

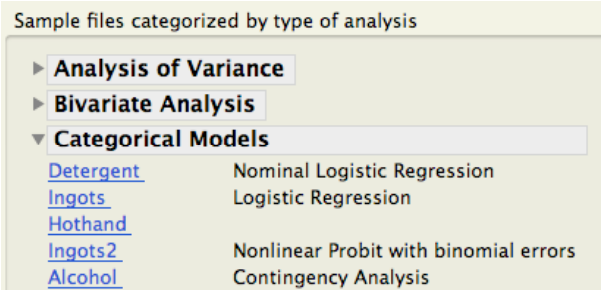
The results are now different.

***Point*: Executing `Eval( x )` is not equivalent to executing `x`.**

3. The Literal Argument Challenge

Suppose you are interested in creating a dialog that contains several outline nodes, each of which contains hyperlinks to different data tables (see Figure 1). The *Sample Data Index* found in the JMP **Help** menu is an example of such a dialog.

Figure 1 Outline Nodes in Sample Data Directory



The following script illustrates how one of these outline nodes could be built, using the JMP sample data index as the example.

```
//Brute Force Method
New Window( "Sample Directory",
Outline Box( "Categorical Models",
Lineup Box( N Col(2), Spacing(0),
Button Box("Detergent",underline style(1),
Open( "$SAMPLE_DATA/Detergent.jmp" )),
Text Box( "Nominal Logistic Regression"),
Button Box( "Ingots2", underline style(1),
Open( "$SAMPLE_DATA/Ingots2.jmp" )),
Text Box( "Logistic Regression" ),
)));
```

This approach rapidly becomes unwieldy when adding statements to construct more and more outline nodes, each with multiple buttons. Instead, imagine a different approach where the script iterates over a list of outline node titles, data table names, and descriptions. As it iterates over the list, it constructs the corresponding dialog.

The following list of lists is for a two-node dialog.

```
// create a list of lists
sample = {
{"Anova",
   {"Blood Pressure", "Multiple Repeated
   Measures"},
   {"Typing Data", "1-way Anova"}
},
{"Categorical Models",
   {"Detergent", "Logistic Regression"},
   {"Ingots2", "Nonlinear Probit Analysis"}}
};
```

Notice that each inner list consists of an initial entry, which is the outline node title. It is followed by several lists of pairs, where the first item is the data table name, and the second item is a table description.

The following function builds the sample file dialog. The `addnode()` function takes two arguments: the first is a reference to a dialog box, and the second is a list. The `addnode()` function is a nested loop that iterates through the list and creates an outline node from the first entry in each inner list. For each inner list pair, it creates a button box with an associated `open()` script, along with the text box that provides the button description. .

```
//Function to build sample file dialog
addnode = Function( {ref, lst},
   For( x = 1, x <= N Items( lst ), x++,
   ref << append( Outline Box( lst[x][1],
   lbx = Lineup Box( N Col( 2 ),
   spacing( 0 )) ) );
   For( y = 2, y <= N Items( lst[x] ), y++,
     table = "$SAMPLE_DATA/" ||
        lst[x][y][1] || ".jmp";
        cmd = Expr( lbx << append( bbx =
        Button Box( lst[x][y][1],
           Open( Expr( table ) ) ) )
        );
           Eval( EvalExpr( cmd ) );
        bbx << underlinestyle;
        lbx << append( Text Box( lst[x][y][2] )
   );
   );
   )
);
```

To start, you need to first create a skeleton dialog to contain the outline nodes and then `addnode()` is invoked.

```
//Create a panel box to contain nodes
New Window( "Sample Files",
   pbx = Panel Box( "Files categorized by
   analysis" ));
//Invoke Sample file function
addnode( pbx, sample );
```

Note that the `append(Button Box(...))` message has been cast as an expression, and that this expression contains a sub-expression, `Expr(table)`. When `Eval Expr( cmd )` is evaluated, `Expr(table)` is replaced with its value and, as a result, the value of `table` at the time of button creation is preserved.

The 'literal argument challenge' in this script occurs in the way the `append(Button Box(...))` message is written. A common mistake is to write the statement thus:

```
lbx << append( bbx = Button Box( lst[x][y][1],
   Open( table ) ) );
```

instead of the correct expression in the script,

```
cmd = Expr( lbx << append( bbx = Button Box(
   lst[x][y][1],
   Open( Expr( table ))))));
```

Although the first statement appears to work, each button actually opens the same data table. In fact, that button always open `Ingots2.jmp`, which happens to be the last data table in the example list. The problem is the `table` variable providing the name for each button. Although `table` contains the correct data table name when each button is created, its value after the dialog is created, and therefore when any button is clicked, will be the last value that was assigned to it.

Here is another correct option.

```
Eval( Substitute(
   Expr( lbx << append( bbx = Button Box( lst[x][y][1],
   Open( xxx ) ) ) ),
   Expr( xxx ), NameExpr( table )));
```

For this solution, the `append(Button Box(...))` message has also been cast as an expression, but it is used here as the first argument of `Substitute()`. Recall that `Substitute()` evaluates its arguments and `NameExpr()` returns the value of its argument unevaluated. So, each time this statement is executed, `Substitute()` returns the value of its first argument but with the value of `table` in place of the pattern *xxx*. Therefore, the effect is the same as the correct solution shown previously.

Concluding Comments

The primary purpose of these examples is to illustrate the use of several expression-handling functions. A secondary purpose is to point out common errors and misunderstandings that JSL programmers sometimes experience when attempting to use these functions. Hopefully, we have partly achieved that objective.

Reference

SAS Institute, Inc. (2008), JMP Scripting Guide, Cary, NC: SAS Institute, Inc.

Sebesta, Robert M. (1999), Concepts of Programming Languages, Addison Wesley, Reading, MA.

Summary Points

Point 3:

Remember that `EvalExpr()` does not evaluate its argument. It clones its argument and replaces any `Expr()` sub-expressions with their evaluated values. Consider this example.

```
y = Expr (
   Distribution(
      Column(Expr("X" || Char(i)))
   )
);
i=3;
x = NameExpr(y);
EvalExpr(x);
```

As expected, statement 4 returns

```
Distribution(Column( "X3" )).
```

So, why not combine statement 3 and statement 4? That is, replace the two separate statements with:

```
EvalExpr(NameExpr(y));
```

When this combined expression executes, `NameExpr(y)` is returned. Note that `EvalExpr()` does not evaluate `NameExpr(y)`; it simply clones it and, since `NameExpr(y)` does not itself contain `Expr()` sub-expressions, `NameExpr(y)` is returned as is.

Point: **`EvalExpr()` does not evaluate its argument.**

Point 4:

If you choose to nest `Eval()` functions, think carefully about how the combined statement will be evaluated. Since `Eval()` evaluates its argument and then evaluates the result, nesting `n Eval()` statements is not equivalent to n instances of `Eval()`. Consider the following example.

```
x = Expr(Expr(Expr(Expr(1 + 2
    ))));
Eval(Eval(x));
y = Eval(x);
Eval(y);
```

Try these statements yourself, executing them one by one, and note the results in the log.

Point: **n nested `Eval()` statements is not equivalent to n `Eval()` statements.**