Developing Formula Evaluation

by Amol Deshmukh

1. Introduction

This document is for developers wishing to contribute to the FormulaEvaluator API functionality.

Currently, contribution is desired for implementing the standard MS excel functions. Place holder classes for these have been created, contributors only need to insert implementation for the individual "evaluate()" methods that do the actual evaluation.

2. Overview of FormulaEvaluator

Briefly, a formula string (along with the sheet and workbook that form the context in which the formula is evaluated) is first parsed into RPN tokens using the FormulaParser class in POI-HSSF main. (If you dont know what RPN tokens are, now is a good time to read this.)

2.1. The big picture

RPN tokens are mapped to Eval classes. (Class hierarchy for the Evals is best understood if you view the class diagram in a class diagram viewer.) Depending on the type of RPN token (also called as Ptgs henceforth since that is what the FormulaParser calls the classes) a specific type of Eval wrapper is constructed to wrap the RPN token and is pushed on the stack.... UNLESS the Ptg is an OperationPtg. If it is an OperationPtg, an OperationEval instance is created for the specific type of OperationPtg. And depending on how many operands it takes, that many Evals are popped of the stack and passed in an array to the OperationEval instance's evaluate method which returns an Eval of subtype ValueEval.Thus an operation in the formula is evaluated.

> **Note:**
> An Eval is of subinterface ValueEval or OperationEval. Operands are always ValueEvals, Operations are always OperationEvals.

`OperationEval.evaluate(Eval[])` returns an Eval which is supposed to be of type ValueEval (actually since ValueEval is an interface, the return value is instance of one of the

implementations of ValueEval). The valueEval resulting from evaluate() is pushed on the stack and the next RPN token is evaluated.... this continues till eventually there are no more RPN tokens at which point, if the formula string was correctly parsed, there should be just one Eval on the stack - which contains the result of evaluating the formula.

Ofcourse I glossed over the details of how AreaPtg and ReferencePtg are handled a little differently, but the code should be self explanatory for that. Very briefly, the cells included in AreaPtg and RefPtg are examined and their values are populated in individual ValueEval objects which are set into the AreaEval and RefEval (ok, since AreaEval and RefEval are interfaces, the implementations of AreaEval and RefEval - but you'll figure all that out from the code)

OperationEvals for the standard operators have been implemented and tested.

2.2. FunctionEval and FuncVarEval

FunctionEval is an abstract super class of FuncVarEval. The reason for this is that in the FormulaParser Ptg classes, there are two Ptgs, FuncPtg and FuncVarPtg. In my tests, I did not see FuncPtg being used so there is no corresponding FuncEval right now. But in case the need arises for a FuncVal class, FuncEval and FuncVarEval need to be isolated with a common interface/abstract class, hence FunctionEval.

FunctionEval also contains the mapping of which function class maps to which function index. This mapping has been done for all the functions, so all you really have to do is implement the evaluate method in the function class that has not already been implemented. The Function indexes are defined in AbstractFunctionPtg class in POI main.

3. Walkthrough of an "evaluate()" implementation.

So here is the fun part - lets walk through the implementation of the excel function... **SQRT()**

3.1. The Code

```
public class Sqrt extends NumericFunction {

    private static final ValueEvalToNumericXlator NUM_XLATOR =
        new ValueEvalToNumericXlator((short)
                ( ValueEvalToNumericXlator.BOOL_IS_PARSED
                | ValueEvalToNumericXlator.EVALUATED_REF_BOOL_IS_PARSED
                | ValueEvalToNumericXlator.EVALUATED_REF_STRING_IS_PARSED
                | ValueEvalToNumericXlator.REF_BOOL_IS_PARSED
                | ValueEvalToNumericXlator.STRING_IS_PARSED
                ));
```

```
    protected ValueEvalToNumericXlator getXlator() {
        return NUM_XLATOR;
    }

    public Eval evaluate(Eval[] operands, int srcRow, short srcCol) {
        double d = 0;
        ValueEval retval = null;

        switch (operands.length) {
        default:
            retval = ErrorEval.VALUE_INVALID;
            break;
        case 1:
            ValueEval ve = singleOperandEvaluate(operands[0], srcRow, srcCol);
            if (ve instanceof NumericValueEval) {
                NumericValueEval ne = (NumericValueEval) ve;
                d = ne.getNumberValue();
            }
            else if (ve instanceof BlankEval) {
                // do nothing
            }
            else {
                retval = ErrorEval.NUM_ERROR;
            }
        }

        if (retval == null) {
            d = Math.sqrt(d);
            retval = (Double.isNaN(d)) ? (ValueEval) ErrorEval.VALUE_INVALID : new Numb
        }
        return retval;
    }

}
```

3.2. Implementation Details

- The first thing to realise is that classes already exist, even for functions that are not yet implemented. Just that they extend from DefaultFunctionImpl whose behaviour is to return an ErrorEval.FUNCTION_NOT_IMPLEMENTED value.
- In order to implement SQRT(..), we need to: a. Extend from the correct Abstract super class; b. implement the evaluate(..) method
- Hence we extend SQRT(..) from the predefined class NumericFunction
- Since SQRT(..) takes a single argument, we verify the length of the operands array else set the return value to ErrorEval.VALUE_INVALID
- Next we normalize each operand to a limited set of ValueEval subtypes, specifically, we call the function `singleOperandEvaluate(..)` to do conversions of different value eval types to one of: NumericValueEval, BlankEval and ErrorEval. The conversion logic is configured by a ValueEvalToNumericXlator instance which is returned by the

Factory method: `getXlator(..)` The flags used to create the ValueEvalToNumericXlator instance are briefly explained as follows: BOOL_IS_PARSED means whether this function treats Boolean values as 1, REF_BOOL_IS_PARSED means whether Boolean values in cell references are parsed or not. So also, EVALUATED_REF_BOOL_IS_PARSED means if the operand was a RefEval that was assigned a Boolean value as a result of evaluation of the formula that it contained. eg. SQRT(TRUE) returns 1: This means BOOL_IS_PARSED should be set. SQRT(A1) returns 1 when A1 has TRUE: This means REF_BOOL_IS_PARSED should be set. SQRT(A1) returns 1 when A1 has a formula that evaluates to TRUE: This means EVALUATED_REF_BOOL_IS_PARSED should be set. If the flag is not set for a particular case, that case is ignored (treated as if the cell is blank) _unless_ there is a flag like: STRING_IS_INVALID_VALUE (which means that Strings should be treated as resulting in VALUE_INVALID ErrorEval)

- Next perform the appropriate Math function on the double value (if an error didnt occur already).
- Finally before returning the NumberEval wrapping the double value that you computed, do one final check to see if the double is a NaN, (or if it is "Infinite") If it is return the appropriate ErrorEval instance. Note: The OpenOffice.org error codes should NOT be preferred. Instead use the excel specific error codes like VALUE_INVALID, NUM_ERROR, DIV_ZERO etc. (Thanks to Avik for bringing this issue up early!) The Oo.o ErrorCodes will be removed (if they havent already been :)

3.3. Modelling Excel Semantics

Strings are ignored. Booleans are ignored!!!. Actually here's the info on Bools: if you have formula: "=TRUE+1", it evaluates to 2. So also, when you use TRUE like this: "=SUM(1,TRUE)", you see the result is: 2. So TRUE means 1 when doing numeric calculations, right? Wrong! Because when you use TRUE in referenced cells with arithmetic functions, it evaluates to blank - meaning it is not evaluated - as if it was string or a blank cell. eg. "=SUM(1,A1)" when A1 is TRUE evaluates to 1. This behaviour changes depending on which function you are using. eg. SQRT(..) that was described earlier treats a TRUE as 1 in all cases. This is why the configurable ValueEvalToNumericXlator class had to be written.

Note that when you are extending from an abstract function class like NumericFunction (rather than implementing the interface o.a.p.hssf.record.formula.eval.Function directly) you can use the utility methods in the super class - singleOperandEvaluate(..) - to quickly reduce the different ValueEval subtypes to a small set of possible types. However when implemenitng the Function interface directly, you will have to handle the possiblity of all different ValueEval subtypes being sent in as 'operands'. (Hard to put this in word, please have a look at the code for NumericFunction for an example of how/why different ValueEvals need to be handled)

4. Testing Framework

Automated testing of the implemented Function is easy. The source code for this is in the file: o.a.p.h.record.formula.GenericFormulaTestCase.java This class has a reference to the test xls file (not /a/ test xls, /the/ test xls :) which may need to be changed for your environment. Once you do that, in the test xls, locate the entry for the function that you have implemented and enter different tests in a cell in the FORMULA row. Then copy the "value of" the formula that you entered in the cell just below it (this is easily done in excel as: [copy the formula cell] > [go to cell below] > Edit > Paste Special > Values > "ok"). You can enter multiple such formulas and paste their values in the cell below and the test framework will automatically test if the formula evaluation matches the expected value (Again, hard to put in words, so if you will, please take time to quickly look at the code and the currently entered tests in the patch attachment "FormulaEvalTestData.xls" file).