MEP Functions

MEP functions can be used to perform arbitrary things with your data (e.g., mathematic operations, string operations, etc.). These functions can be used in different operators like Map, Select, or Join. To implement a MEP function, one has to extend the *AbstractFunction* class. To implement your own MEP function you basically have to implement the **getValue** function that calculates the return value and call the super constructor with the configuration of your MEP function. The configuration contains at least the symbol, the number of parameters, the accepted data types for the parameters, and the data type of the return value. In addition, the configuration can contain a flag to indicate if the MEP function should be evaluated each time or if it is a constant and the time and space complexity of the MEP function.

MEP Function stub

```
public class MyFunction extends AbstractFunction<Double> {

    public static final SDFDatatype[][] accTypes = new SDFDatatype[][] {{ SDFDatatype.DOUBLE },
                                                                        { SDFDatatype.DOUBLE }};

    public MyFunction() {
        super("myFunction", 2, accTypes, SDFDatatype.DOUBLE, true, 3, 5);
    }

    @Override
    public Double getValue() {
        double a = (double) this.getInputValue(0);
        double b = this.getNumericalInputValue(1);

        return a  +b;
    }

 }
```

In this example a MEP function is defined with the symbol *myFunction* and can be used in a predicate or map expression with two parameters of type Double and will also return a value of type Double which will be the sum or the two input parameters as defined by the implementation of the *getValue()* function. Further, the MEP function can be optimized if the two input parameters are constant. Then, the result will be calculated only once. In addition, the MEP function provides a time complexity score of *3* and a space complexity score of *5* which means that the optimizer will try to place the call of *myFunction* before any other MEP function with a higher time or space complexity and behind any MEP function with a lower time or space complexity during optimization.

Access to function attributes

To Access the attributes of the function you can use the *getInputValue* or the *getNumericalInputValue* methods. While the first method returns an object, the second already cast the input value to a double value. Both methods takes the position index of the attribute as an argument. The name of the function, the total number of attributes, and the data type of the accepted attributes is set in the *constructor*. Thus, a MEP function can handle multiple data types for each attribute.

Access to meta attributes

To access the meta attributes of an incoming streaming object you can use the *getMetaAttribute* function.

Access to additional content

To access the additional content of an incoming streaming object you can use the *getAdditionalContents* method to access all contents. If you only want to access a special field you can issue the *getAdditionalContent(fieldName)* method.

Support for optimization

The MEP optimizer tries to determine if an expression is a constant and should not be evaluated each time. For this, the getValue method is called. This behavior can be changed by setting the fifth parameter in the constructor to false. To support the optimization of predicates, the time and space complexity can be set in the constructor as the last two parameters. Both values should be in the range between 0-9 depending on their average expected complexity. Depending on the value, the MEP function will be placed differently in the resulting optimized predicate.

Image the following scenario with a predicate expression that checks if an attribute x holds a value higher than the return value of a function called *lastPrimeNumber(x)* (that might estimate the last prime number lower or equals to x), a value higher than the result of function defined above, and higher than 0:

(x > lastPrimeNumber(x)) || (x > myFunction(y, z)) || (x > 0)

During predicate optimization, the optimizer checks the complexity values and reorder the terms according to their values resulting in an optimized predicate as follows:

(x > 0) || (x > myFunction(y, z)) || (x > lastPrimeNumber(x))

Here, the last term comparing the value of x with 0 is moved to the front and the more expensive function is moved to the tail.

A rule of thumb should be, MEP functions with logarithmic complexity should have a value between 0-3, linear complexity a value between 4-6, and exponential complexity a value between 7-9. However, this is just a first draft. The basic idea is to evaluate cheap functions first and avoid expensive functions if possible.