Programmer's Guide for New MMDB Operators

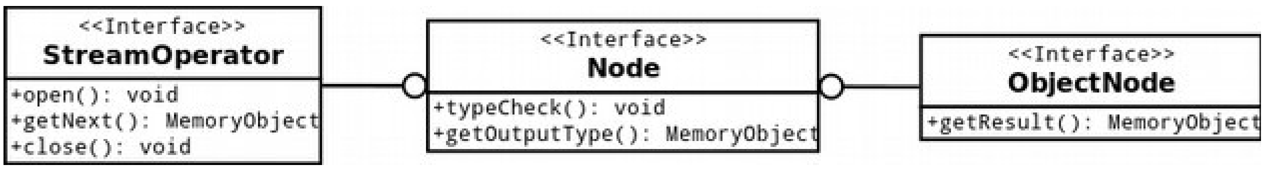
This guide demonstrates how to implement new operators for the main memory query processor of the SECONDO gui. Since the basic design for the stream processing component of the main memory module is mainly based on interfaces, one is quite free in how to design internal parts of an operator. This guide is supposed to give some conventions on how to solve internal tasks of any operator to keep the differences in programming manageable and to ease up maintenance of operators by different authors. Please try to stick to these conventions unless you have a good reason not to, and if so, please leave a comment in the code explaining why it does things differently.

To ease up understanding of certain aspects, this guide contains example code from basic operators from the SECONDO core, which are already implemented in the MMDB. These should give you an impression on how the explained techniques are applied. You will find short explanations for what all example operators do in appendix A.

Furthermore it might be helpful to have the full UML class diagram at hand, that you will find as appendix B.

Output Orientation

First thing to do is to decide whether the operator you want to implement is a StreamOperator - providing a stream of results - or an operator that provides just one result. There are two main interfaces that have to be used for these, called “StreamOperator” and “ObjectNode”. Your new operator class must implement exactly one of these.



Both of these interfaces extend the interface “Node” which represents every type of node in the final tree representing a database query. The tree consists only of Nodes. Constants (e.g. strings, numbers, existing database elements, etc.) are wrapped into ObjectNodes, so an operator does not distinguish between constants and calculated results.

Every StreamOperator supports the iterator interface consisting of the methods open(), getNext() and close(). On open() the operator usually calls open() on its argument Nodes and might initialize some local variables needed to process getNext() calls. GetNext() is the equivalent to an ObjectNode’s getResult() method. It returns the next element of the stream, or null, if there are no more elements. Close() closes its arguments and makes sure the operator is ready to be reopened.

An ObjectNode in contrast only offers the getResult() method, which returns the single result of its calculations.

Plus and Extend make two good example operators here, one returning a single result, and one putting out a stream. Their class definitions look this this:

```
public class Plus implements ObjectNode {
```

```
public class Extend implements StreamOperator {
```

Constructor

After deciding if your operator puts out a stream of results or a single result, the next step is to fix the number and types of arguments it needs. There are two basic types of arguments: Nodes (typed as the interface mentioned above) and identifiers (typed as simple Java strings to separate them from calculated strings or constants). If your operator has a fixed number of arguments, writing its constructor is simple. Here are some basic examples:

```
public Feed(Node input) {
	this.input = input;
}

public Plus(Node input1, Node input2) {
	this.input1 = input1;
	this.input2 = input2;
}

public Rename(Node input, String postfix) {
	this.input = input;
	this.postfix = postfix;
}
```

If your operator supports a variable number of arguments of the same type, just use an array of that type. For instance for the Project operator, supporting a list of identifiers with a variable length, looks like this:

```
public Project(Node input, String[] identifiers) {
	this.input = input;
	this.identifiers = identifiers;
}
```

It gets a little more complex, if your operator is to accept lists of pairs of arguments. If that's the case, the parser provides Java LinkedHashMaps, to keep the order of argument pairs intact. The constructor of the Extend operator which is to accept a list of identifier->ParameterFunction pairs, then looks like this:

```
public Extend(Node input1, Map<String, Node> input2) {
	this.input1 = input1;
	this.input2 = input2;
}
```

The Groupby operator, additionally expecting a list of grouping identifiers, thus has one more parameter:

```
public Groupby(Node input1, String[] identifiers, Map<String, Node> input3) {
	this.input1 = input1;
	this.identifiers = identifiers;
	this.input3 = input3;
}
```

The maps are iterable in their original order, which makes them suitable for this purpose. Needless to say, all arguments should be stored in private member variables for later use.

Regarding the order of your operator's arguments, please stick to the argument order of the operator in the core, if you are rewriting a core operator. The order should always fit the order of arguments of your operators arguments in nested list format, that makes it easier to read. Constructors are called internally by a method called fromNL() also implemented in your operator (see section “Parsing” for details).

Type Checking

After an operator tree has been built and thus your operator’s constructor has already been used, the first thing to do is to perform a type check. Without being type checked, a Node can neither know its output type nor can it guarantee to be able to return any results.

Type checking ensures, that the (sub-)tree is evaluable and that there are no type errors. It is usually done recursively from the root node down to each leaf and each Node has to implement the typeCheck() method to perform a type check on it. Once any error has been found, a TypeException is to be thrown, stopping the typeCheck() process and containing detailed information about the error as its message.

Besides looking for type errors, type checking also calculates the output type of a Node (see section “Output Types”). Thus (and for the purpose of recursion) it is important to start the typeCheck() method by calling typeCheck() on all Node arguments (exceptions: see section “Parameter Functions”). This ensures that all arguments know their output types and that your operator can check if it can work with them, and what its own output type will be. For the Plus operator the implementation thus begins like this:

```
@Override
public void typeCheck() throws TypeException {
	this.input1.typeCheck();
	this.input2.typeCheck();
	...
```

Once a typeCheck() has been performed on all argument Nodes, the next step should be to check if argument Nodes offer the right interfaces. For each argument given in the constructor you should know if it’s supposed to be a StreamOperator, an ObjectNode or a ParamterFunction. Checking can be done using the helper class TypecheckTools. If the used method in TypecheckTools did not throw a TypeException, perform a down cast on the Node and store the result in a new member variable of the type of the interface needed (StreamOperator/ObjectNode/ParameterFunction). For the Plus operator this step looks like this:

```
// Is input1 an ObjectNode?
TypecheckTools.checkNodeType(this.input1, ObjectNode.class,
            this.getClass(), 1);
this.objectInput1 = (ObjectNode) this.input1;

// Is input2 an ObjectNode?
TypecheckTools.checkNodeType(this.input2, ObjectNode.class,
            this.getClass(), 2);
this.objectInput2 = (ObjectNode) this.input2;
...
```

Further steps completely depend on the operator you want to implement. Almost certainly you also need to check the output types of the arguments to ensure your operator can handle them. The Rename operator for instance additionally requires its StreamOperator input to provide tuples:

```
// Is input a StreamOperator providing Tuples?
TypecheckTools.checkNodeType(this.input1, StreamOperator.class,
            this.getClass(), 1);
TypecheckTools.checkOutputType(this.input1, MemoryTuple.class,
            this.getClass(), 1);
this.streamInput = (StreamOperator) this.input1;
```

As you can see, there are helper methods in TypecheckTools as well for this purpose. Finally, never forget to determine and store your own output type for later use in your own getOutputType() method. You will see how the Plus operator does this in an example further below.

If your operator is supposed to handle different input combinations (we use the Plus operator as the example for the rest of this section), evaluation methods should mostly differ for each of these combinations. We suggest handling this the following way: Create a private enum in your operator class and add an understandably named value for each input combination:

```
private enum InputVariant {
      INT_INT, INT_REAL, REAL_INT, REAL_REAL, STRING_STRING
}
```

During the type check you should evaluate which of these combinations is present, and store it in a private instance of this enum. This enum can be used in evaluation phase to determine which evaluation method is to be used. Also your output type can be dependent on the input combination and if so it should be set according to it:

```
this.inputVariant = determineInputVariant(this.objectInput1.getOutputType(),
      this.objectInput2.getOutputType());

private InputVariant determineInputVariant(MemoryObject obj1,
            MemoryObject obj2) throws TypeException {
      if (obj1.getClass() == AttributeInt.class) {
            if (obj2.getClass() == AttributeInt.class) {
                this.outputType = new AttributeInt();
                return InputVariant.INT_INT;
            }
            if (obj2.getClass() == AttributeReal.class) {
                this.outputType = new AttributeReal();
                return InputVariant.INT_REAL;
            }
        }
        ...
        throw new TypeException(
            "%s's %ss provide an unsupported datatype combination: %s + %s.",
            this.getClass().getSimpleName(), Nodes.NodeType.ObjectNode,
            obj1.getClass().getSimpleName(), obj2.getClass()
                    .getSimpleName());
}
```

As you can see, if no valid input combination has been found, a TypeException is thrown informing about that error.

Output Types

Every Node offers a method named getOutputType() to get to know the type of output this node returns. No matter if your operator is a StreamOperator or an ObjectNode, it puts out pieces of data represented as MemoryObjects. A MemoryObject can be e.g. a tuple, a relation, an attribute or another piece of data like an index, etc. If your operator serves as an argument for another operator, this operator has to be able to determine your operator's output type.

For many pieces of data like most attributes for instance, it is enough to know its “type”, which can be represented by its class(-name). For complex objects like e.g. tuples, most of the time it’s not enough to know it’s a tuple, but its composition must be known as well.

To have a unified return value for getOutputType() there are typecheck instances of MemoryObjects. These do not represent the data itself (they usually don’t carry any “content”) but just represent its type. If the object is a simple object, just create an (empty) instance of its type, using its (0 parameter) standard constructor and return it in getOutputType(), as seen in Plus exampe above:

```
this.outputType = new AttributeReal();
```

If it’s a complex object, the MemoryObject’s class should offer a static method called createTypecheckInstance() which has a parameter describing the object’s composition. This returns an empty typecheck instance carrying the composition information. If you are checking the output type of an argument operator (should be *only* during type check) you can use the getTypecheckInfo() method to get exactly this information. Extend does those two steps the following way:

```
private MemoryTuple calculateOutputType() throws TypeException {
      // Retrieve OutputType of streamInput
```

```
        MemoryObject inputTypecheckInstance = this.streamInput.getOutputType();
        List<RelationHeaderItem> inputTypecheckInfo =
                ((MemoryTuple) inputTypecheckInstance).getTypecheckInfo();

        // Create new TypecheckInfo based on input
        List<RelationHeaderItem> newTypecheckInfo =
                new ArrayList<RelationHeaderItem>();
        newTypecheckInfo.addAll(inputTypecheckInfo);

        // Add all new Attributes
        ...
        return MemoryTuple.createTypecheckInstance(newTypecheckInfo);
}
```

For non complex objects just use Java's "instance of" or getClass().getName() to compare types (see Plus example above).

Parameter Functions

Parameter functions are an ability of the SECONDO core and enable an operator to delegate calculations upon each of its stream input's elements to a sub node.

In the Java client's implementation a parameter function is mostly a wrapper for a sub operator tree that is evaluated once for each parameter that your operator sends to it. This sub tree can be of any complexity and its root node might be a StreamOperator or an ObjectNode. On an evaluate() call it gets passed a parameter of type MemoryObject, makes it known to all sub operators accessing this parameter, and then passes the root node of the sub tree to the callee who can retrieve the single or multiple results from it.

Inside the sub tree there may be unlimited operators who need access to the current parameter content. For this purpose there is an ObjectNode implementation called FunctionEnvironment. A ParameterFunction has references to one or multiple FunctionEnvironments (depending on the number of its parameters) and these can be linked into different (and multiple) places in the ParameterFunction's sub tree:

```
public ParameterFunction(Node operator, Node... parameters) {
        this.operator = operator;
        this.parameters = parameters;
}
```

Each time the ParameterFunction receives an evaluate() call, it passes the parameter(s) to the FunctionEnvironment(s) which can then answer their getResult() request(s) correctly:

```
public Node evaluate(MemoryObject... parameters) {
        for (int i = 0; i < parameters.length; i++) {
                this.functionEnvironments[i].setObject(parameters[i]);
        }
        return this.operator;
}
```

What's important for implementation? If your operator uses a ParameterFunction as an argument, this node cannot be type checked recursively right at the beginning of type checking like every other argument. For it to be type checked, it needs to know the type(s) of its parameter(s). Since the parameters are the elements of a StreamOperator argument of your operator, you can ask your other argument(s) for the parameter type(s) and then pass it/them on to the ParameterFunction via setParamTypes(). Once this has been done, the ParameterFunction can be type checked normally. This process is shown here for the Filter operator:

```java
// Typecheck ParameterFunction
this.functionInput.setParamTypes(this.streamInput.getOutputType());
this.functionInput.typeCheck(); // !!! Important !!!
```

After that a ParameterFunction offers two different methods for checking its output. The inherited getOutputType() method returns the output type of the operator returned by the ParameterFunction. A special method getOperatorType() returns the type of operator itself and thus tells you if it's a StreamOperator or an ObjectNode. For all these checks there are helper methods in TypecheckTools again as seen in this Filter example:

```java
// Does the ParameterFunction provide an ObjectNode providing an AttributeBool?
TypecheckTools.checkFunctionOperatorType(this.functionInput,
            ObjectNode.class, this.getClass(), 2);
TypecheckTools.checkFunctionOutputType(this.functionInput,
            AttributeBool.class, this.getClass(), 2);
```

Evaluation Method

The main evaluation methods are getNext() for StreamOperators, getResult() for ObjectNodes and evaluate(MemoryObject...) for ParameterFunctions. Keep in mind, that here you should not need to access any type check info from your arguments, since you should already have checked and stored all you need during type checking. This cleanly separates type checking from evaluation.

As described in the “Type checking” section, if your operator handles different input type combinations and you have initialized an enum for this purpose during type check, you should now use that enum to delegate a getResult() or getNext() call to a specific (private) implementation of the evaluation method inside your operator, that can handle the present input combination. For the Plus operator this is done as follows:

```java
@Override
public MemoryObject getResult() {
      if (this.objectInput1.getResult() == null
                  || this.objectInput2.getResult() == null) {
            return null;
      }

      switch (this.inputVariant) {
      case INT_INT:
            return getIntIntResult();
        case INT_REAL:
            return getIntRealResult(false);
        ...
        default:
            return null;
        }
}
```

Further content of evaluation methods of course depends completely on what your operator is supposed to do. For this reason we're mostly just giving some hints on what to think about:

- If you have StreamOperator arguments:
    - How and when do you react to the stream's end (null value returned)?
    - Can you handle a stream that is empty from the start?
- If you have ObjectNode arguments:
    - How do you react, if the getResult() method returns null (representing “undefined”)? Always check the core's resembling operator's reaction to these situations first.
- If you implement a StreamOperator:
    - When do you return null/has your stream ended?
    - After once returning null, your operator should always return null until it's close()-ed.
    - How do you react, if the evaluation method has been called without open() being called beforehand? There's “StreamStateException” for this purpose to be thrown.
- If you implement an ObjectNode:
    - In which cases is your result undefined? Return null then.

Memory Management

Since the mmdb query processor naturally stores all results in main memory there might be cases when memory tends to run out during evaluation of certain queries. Assuming the query's output is a relation, every tuple that's being gathered in the root (Consume) Node adds to the total memory being in use. With large result relations (millions of tuples) this memory usage can easily go up into the hundreds of megabytes.

To handle these situations where usually an OutOfMemoryError might occur, the author of the storage management for the MMDB delevoped a class called MemoryWather that can be called to check if enough free memory is left at the time. It offers a method called checkMemoryStatus() that checks if a configured percentage of main memory for the JVM (standard: 20%) is still free and throws a MemoryException if it's not. These MemoryExceptions are caught in the ParserController and thus lead to a controlled abortion of query procession. The user is informed that not enough memory was left for procession and that either some memory objects have to be removed beforehand or that the JVM has to be started with more memory allocated to it.

Your operator should call the checkMemoryStatus() method whenever it locally stores a possibly huge (or unlimited) amount of stream elements. Since checking memory after every gathered element would be very expensive, there also is a frequency configured in the MemoryWatcher (standard: every 100th tuple). Using a counter variable in your operator's code memory checking may thus look like this (taken from Consume operator):

```
int memorywatch_counter = 0;
while (...) {
	memorywatch_counter++;
	if (memorywatch_counter % MemoryWatcher.MEMORY_CHECK_FREQUENCY == 0) {
		MemoryWatcher.getInstance().checkMemoryStatus();
	}
	tuples.add(tuple);
}
```

Now after every 100th gathered element of the stream the MemoryWatcher checks if enough free memory is left.

The clearest example of course is the Consume operator that gathers all result tuples of a stream. In its getResult() method memory thus is checked like seen above and a MemoryException may be thrown. So for ObjectNodes in general the getResult() method may throw MemoryExceptions. For StreamOperators you might do a lot of work already on the call to your open() method since you might need to initialize temporary tables, etc. An example for this is the Hashjoin operator: It stores one of it's incoming streams entirely inside a hashmap. Since you might also run out of memory during the getNext() method (e.g. while calling a parameter ObjectNode's getResult() method), both of these methods (open(), getNext()) may throw MemoryExceptions in StreamOperators.

Please always consider if your operator temporarily stores data and if so, implement the memory check similar to the way seen above.

Parsing

Since the SECONDO query language is context dependent, your operator has to provide some information about what its arguments have to be like (i.e. what types they must have). Your constructor already contains this information, but since the parser does not make heavy use of reflection, your operator has to be able to handle it's arguments in another way.

A SECONDO query in the core is at first parsed into a nested list format (if it was not entered in that format) and the operator tree is built up based upon the nested list representation of the query. Every operator in the core has a function called “typemap” that checks its arguments based on their nested list representation.

For the MMDB query processor a similar way has been chosen. So before building up the operator tree the parser stores the query in a nested list format. The transformation to this format (if needed) is done by the core. Every operator has to implement the method fromNL() (from nested list) with a fixed signature that the parser calls via reflection (yes, this call is one of the very few uses of reflection in the query processor). The signature looks as follows:

```
public static Node fromNL(ListExpr[] params, Environment environment);
```

The second parameter is a pass through object and it is unlikely, that you will actively use it. The „Environment“ contains all named objects in the database as well as objects that have been defined during the execution of the current query. So if your operator does not define any named objects for its subtree (like parameter functions do for instance) you will only have to pass it through. For the curious reader: The Environment object easily implements a basic scoping technique by making its internal list unmodifiable. Thus if any objects are added to the environment, they can only be seen in your operator's parameters (subtree), since any parent nodes still are linked to their old environment instance.

The first parameter is the important one: it contains all the parameters your operator has gotten in the current query in an internal nested list format. Every one of these can be a list or an atom (like usual in nested lists). The implementation of the fromNL() method now has to make sure that these are of the right type and number. In most cases this is very simple, since the class NestedListProcessor offers many static helper methods that operators use for this purpose. They all are named “nlTo...” and return the requested object based on a nested list.

For the plus operator for instance the fromNL() method looks like this:

```
public static Node fromNL(ListExpr[] params, Environment environment)
          throws ParsingException {
     ParserTools.checkListElemCount(params, 2, Plus.class);
     Node node1 = NestedListProcessor.nlToNode(params[0], environment);
     Node node2 = NestedListProcessor.nlToNode(params[1], environment);
     return new Plus(node1, node2);
}
```

At first it checks the number of parameters. For plus this is two, and for most operators it is fixed, since any parameter types that can occur in variable numbers are usually packed into one nested list and thus only count as one parameter (see next example). After making the important calls to nlToNode() which request the NestedListProcessor to produce Nodes out of nested lists, these nodes are passed on to Plus' own constructor and the new Plus instance is returned.

This process can be seen as semi recursive, since Plus' parameters can be Plus nodes as well (or of any other subtype of Node). So this is repeated through to the operator tree's leafs and then the tree is built from leafs to the root.

Most operators use a basic set of parameter types, so the NestedListProcessor offers “constructors” for all of them. The Groupby operator e.g. has a node parameter, a list of grouping identifiers and a list of pairs of identifiers and nodes. It's fromlNL() method still is not really more complicated:

```
public static Node fromNL(ListExpr[] params, Environment environment)
          throws ParsingException {
     ParserTools.checkListElemCount(params, 3, Groupby.class);
     Node node1 = NestedListProcessor.nlToNode(params[0], environment);
     String[] identList = NestedListProcessor.nlToIdentifierArray(params[1]);
     Map<String, Node> paramMap = NestedListProcessor
```

```
                .nlToIdentifierNodePairs(params[2], environment);
        return new Groupby(node1, identList, paramMap);
}
```

See NestedListProcessor's public interface for more details.

In addition to implementing the fromNL() method you only have to register your operator in the OperatorLookup class. It is located under mmdb.streamprocessing.parser and contains a HashMap of all available operators. Just add a line like this:

```
operatorMap.put("feed", Feed.class);
```

in the initializeMap() method and you are done!

Testing

Last but not least all operators should have their own JUnit test classes which test different scenarios the operator could occur in. Use a framework like EclEmma to check code coverage. All public interface methods of you operator should be well tested, so your code coverage should be > 90%. After finishing you test class, do not forget to add it to the streamprocessing TestSuite under test/unittests.mmdb.suites ("TestSuiteStreamProcessing"). This ensures it is executed while performing an overall test run for all JUnit test cases.

To create a testing environment, built up the arguments you need using ConstantNodes and then assemble your operator. A simple test method for the Plus operator looks like this:

```
@Test
public void testIntInt() throws TypeException {
        ObjectNode attrNode1 = ConstantNode.createConstantNode(
                new AttributeInt(5), new AttributeInt());
        ObjectNode attrNode2 = ConstantNode.createConstantNode(
                new AttributeInt(7), new AttributeInt());
        Plus plus = new Plus(attrNode1, attrNode2);
        plus.typeCheck();

        assertEquals(AttributeInt.class, plus.getOutputType().getClass());
        assertEquals(12, ((AttributeInt) plus.getResult()).getValue());
}
```

Do not forget to typecheck() your operator tree's root before pulling results. If your operator is a StreamOperator keep in mind that you also need to call open() before getNext(), otherwise a StreamStateException should be thrown. Here's a simple test for the Feed operator:

```
@Test
public void testFeed() throws TypeException {
        MemoryRelation rel = TestUtilRelation.getIntStringRelation(5, true,
                false);
        ObjectNode ObjectNode = ConstantNode.createConstantNode(rel, rel);
        Feed feed = new Feed(ObjectNode);
        feed.typeCheck();
        feed.open();
        for (int i = 0; i < 5; i++) {
              assertNotNull(feed.getNext());
        }
        assertNull(feed.getNext());
        assertNull(feed.getNext());
        feed.close();
}
```

As you can see here, there are helper methods available to create certain MemoryRelations in TestUtilRelation.

In terms of special cases to test review the list of tips in the evaluation method section. All these questions/hints should be tested to ensure your operator works correctly also under non standard conditions.

Additionally for each operator there should be at least one testQuery() test that tests a query in nested list format. Just place this query in a String and forward it to the NestedListProcessor. An example for Plus follows:

```
@Test

public void testQuery() throws Exception {
        String query = "(query (+ 3 4))";
        ObjectNode result = NestedListProcessor.buildOperatorTree(query,
                    new ArrayList<SecondoObject>());
        result.typeCheck();
        assertEquals(new AttributeInt(7), result.getResult());
}
```

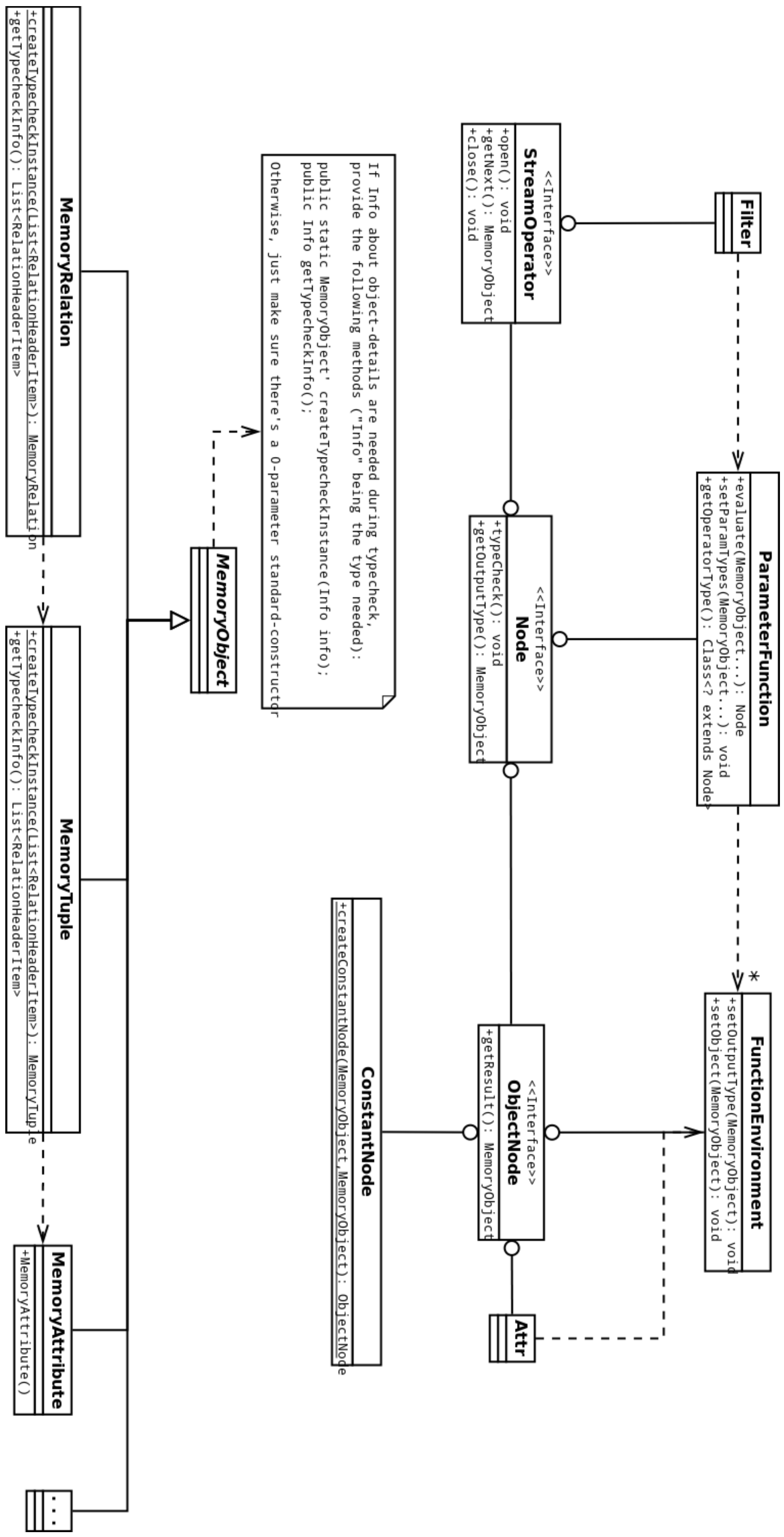
As you can see the resulting ObjectNode is typechecked and it's result is compared with the desired result. The primary goal of this is to ensure that your operator can be built from a nested list representation, the secondary goal is of course, that it works correctly. Both can be ensured in mostly very easy tests.

Appendix A: Example Operators

The following list contains all example operators in the order they have been used in the guide and gives brief descriptions of their functionality.

- Plus: Simple “+” Operator, adding two figures or concatenating two strings
- Extend: Offers the “SELECT… AS…” features of SQL. Takes a stream of tuples and calculates new attributes based on the existing ones
- Feed: Takes a relation and offers a stream of its tuples
- Rename: Takes a stream of tuples and appends a postfix to all column identifiers
- Project: A simple projection operator. Works like the SQL SELECT clause and thus filters columns
- Filter: A StreamOperator realising the SQL WHERE clause. Takes MemoryTuples and filters them using a ParameterFunction
- Groupby: A StreamOperator offering the functionality of the SQL “group by” feature. It only works on sorted streams though.

Appendix B: UML Class Diagram


MemoryRelation
+createTypecheckInstance(List<RelationHeaderItem>): MemoryRelation
+getTypecheckInfo(): List<RelationHeaderItem>
MemoryTuple
+createTypecheckInstance(List<RelationHeaderItem>): MemoryTuple
+getTypecheckInfo(): List<RelationHeaderItem>
MemoryAttribute
+MemoryAttribute()
...
MemoryObject
<<Interface>>
StreamOperator
+open(): void
+getNext(): MemoryObject
+close(): void
Filter
<<Interface>>
Node
+typeCheck(): void
+getOutputType(): MemoryObject
ParameterFunction
+evaluate(MemoryObject...): Node
+setParamTypes(MemoryObject...): void
+getOperatorType(): Class<? extends Node>
ConstantNode
+createConstantNode(MemoryObject,MemoryObject): ObjectNode
<<Interface>>
ObjectNode
+getResult(): MemoryObject
Attr
*
FunctionEnvironment
+setOutputType(MemoryObject): void
+setObject(MemoryObject): void

```
If Info about object-details are needed during typecheck,
provide the following methods ("Info" being the type needed):
public static MemoryObject' createTypecheckInstance(Info info);
public Info getTypecheckInfo();
Otherwise, just make sure there's a 0-parameter standard-constructor
```