Assingment 2 - External DSL Interpreter

Marc Bertelsen
berte20@student.sdu.dk

March 27, 2024

1 Summary

- XText grammer - done
- XTend generator - done
- XTend validator - done
- XTend scoping - todo
- Tests
  - MathExampleTest - all pass
  - MathParsingTest - all pass
  - MathScopeTest - error
  - MathValidatorTest - all pas

2 Desing

Important for the design of the metamodel and syntax, is the enforcement of mathematical operation execution. This must be implemented with the already given syntax from the test cases.

2.1 Metamodel

Fig. 1, show a simple metamodel for the design of a math interpreter. By structuring the model such that the different expression sub-types share a common base class, it enables the usage of a single function for the expression computations. Additionally, the structure on the metamodel enforces the correct order of mathematical operations, by the way the expression tree is composed. Addition and subtraction at the top, this will happen last in the order of operation. Multiplication and division are below, which means they will execute one step before. All the Primary sub-types require even earlier computation, so they are placed deeper in the tree.

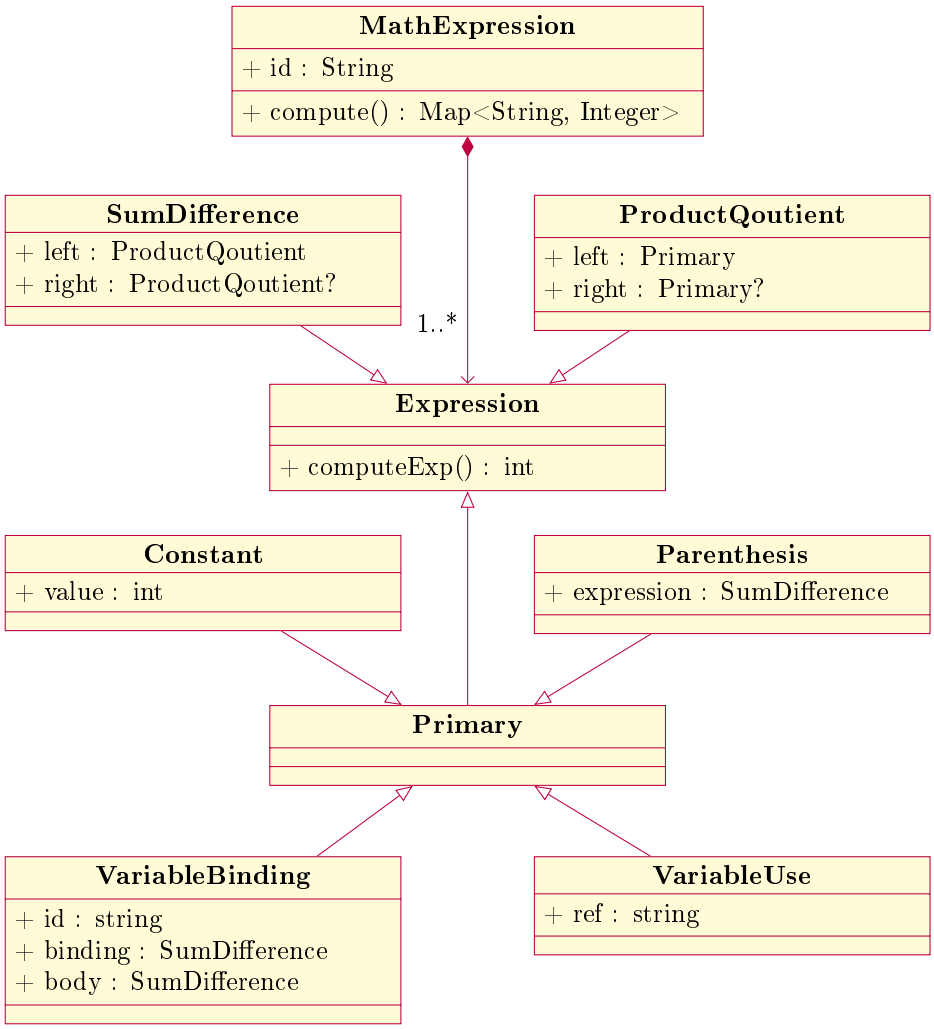

Figure 1: Mathematical Expression Metamodel

2.2 Syntax

The syntax is already given in the different test cases. So, the resulting *XText* grammar has to accommodate syntax as seen in fig. 1

```
var a = 1
var b = 1 + 1
var c = a + b
var d = let x = 1 + 1 in a end
```

Listing 1: Examples of *.math syntax

3 Implmentation

Repo: `https://github.com/Waf197/MDSD/tree/main/A2/assignment2`

3.1 XText syntax

```
grammar dk.sdu.mmmi.mdsd.Math with org.eclipse.xtext.common.Terminals

generate math "http://www.sdu.dk/mmmi/mdsd/Math"

MathExp:
    exps+=Exp*
;

Exp:
    'var' name=ID '=' exp=SumDiff
;

SumDiff returns Expression:
    ProdQuot (('+'{Add.left=current} | '-'{Sub.left=current}) right=ProdQuot)*
;

ProdQuot returns Expression:
    Primary (('*'{Mul.left=current} | '/'{Div.left=current}) right=Primary)*
;

Primary returns Expression:
    Constant | Parenthesis | VariableUse | VariableBinding
;

Parenthesis returns Expression:
    {Parenthesis} '(' exp=SumDiff ')'
;

Constant returns Expression:
    {Constant} value=INT
;

VariableUse returns Expression:
    {VariableUse} ref=ID
;

VariableBinding returns Expression:
    {VariableBinding} 'let' id=ID '=' binding=SumDiff 'in' body=SumDiff 'end'
```

;

Listing 2: XText syntax

3.2 XTend Generator

For solving the *evilExample* test, a map for getting variables, that is yet to be computed, was used. This way if the variableBinding can get variables before they are computed and compute its associated expression.
Since all the sub-expression types are all derived from the same super class (**Expression**) it is possible to use the same function (**computeExp**) to compute the value based on the sub-type.

```
override void doGenerate(
    Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
    val variables = resource.allContents.filter(MathExp).next.compute

    // You can replace with hovering, see Bettini Chapter 8
    variables.displayPanel
}

def static Map<String, Integer> compute(MathExp math) {
    val variables = new HashMap<String, Integer>()

    val fwExp = new HashMap<String, Expression>()
    math.exps.forEach[exp | fwExp.put(exp.name, exp.exp)]

    math.exps.forEach[exp | {
        val res = exp.exp.computeExp(variables, fwExp)
        variables.put(exp.name, res)
    }]
    return variables
}

def static int computeExp(
    Expression exp, Map<String, Integer> vars, Map<String, Expression> fwExp) {
    switch exp {
        Add: exp.left.computeExp(vars, fwExp)+exp.right.computeExp(vars, fwExp)
        Sub: exp.left.computeExp(vars, fwExp)-exp.right.computeExp(vars, fwExp)
        Mul: exp.left.computeExp(vars, fwExp)*exp.right.computeExp(vars, fwExp)
        Div: exp.left.computeExp(vars, fwExp)/exp.right.computeExp(vars, fwExp)
        Constant: exp.value
        Parenthesis: exp.exp.computeExp(vars, fwExp)
        VariableUse:
        {
            if (!vars.keySet.contains(exp.ref)) {
```

```
                val res = fwExp.get(exp.ref).computeExp(vars, fwExp)
                vars.put(exp.ref, res)
            }
            vars.get(exp.ref)
        }
        VariableBinding: exp.body.computeExp(
            vars.bind(exp.id, exp.binding.computeExp(vars, fwExp)), fwExp)
        default: throw new Error("Could not compute expression")
    }
}

def static Map<String, Integer> bind(
    Map<String, Integer> vars, String key, Integer value) {
    val binding = new HashMap<String, Integer>(vars)
    binding.put(key, value)
    binding
}
```

Listing 3: XTend generator

3.3 XTend Validator

Checking for duplicate var declarations. This is done with a Set, that is cleared when checking the **MathExp** since it is the first thing that is checked. Hereafter each of the **Exps** are checked to see if they are in the Set, if yes, then issue a warning and return, if no, add it to the set.

```
public static val DUBLICATE_VAR = 'dublicateVar'

val set = new HashSet<String>()

@Check
def clearSet(MathExp m) {
    set.clear
}

@Check
def checkNoDublicateVar(Exp exp) {
    if (set.contains(exp.name)) {
        warning("var " + exp.name + " has already been declared",
            MathPackage.Literals.EXP__NAME,
            DUBLICATE_VAR
        )
        return
    }
    set.add(exp.name)
```

}

Listing 4: XTend validator

4 Test

Current implmentation passes **MathExampleTest**, **MathParsingTest**, and **MathValidatorTest**, but failes **MathScopeTest**.

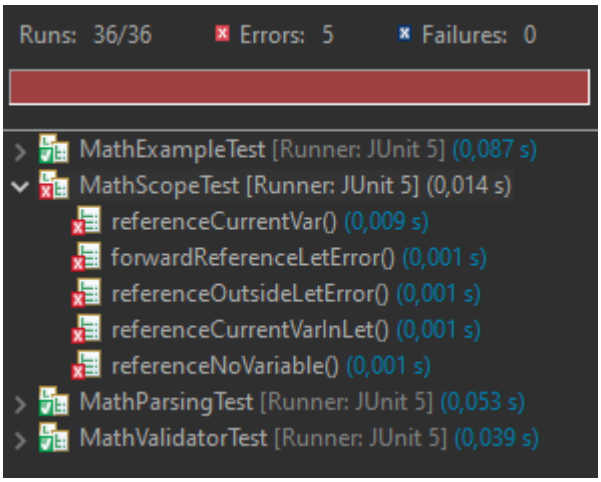

Figure 2: Test Results

5 Conclusion

The current implementation solves most of the test, however there might be a more elegant solution for solving the *evilExample* test, by using better **XText** grammar. How it is solved currently feels like a bit of a hack.