The project passes all the tests.
It is implemented by never referencing a a variable, but always walking through the execution path to build the formulas. This is done to avoid issues with the order the variables are written. This results in something that is functionally equivalent, as long as the code is only generated an never manually changed.

The validator validates correct parameter count of external use, and it validates duplicate names as with the last assignment.

The scoping uses nested scopes to best handle shadowing

Repository: https://github.com/Hounsvad/Sem8/tree/master/ModelDriven/assignment3

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

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

Program:
	programName=ProgramName externals+=External* variableAssignments+=VariableAssignment+
;

ProgramName returns ProgramName:
	'program' name=ID
;

External:
	{External} 'external' name=ID '(' parameters+=ID? (',' parameters+=ID)* ')'
;

ExternalUse returns Expression:
	{ExternalUse} ref=[External] '(' exp+=Exp? (',' exp+=Exp)* ')'
;

//Variables:
//	variableAssignments+=VariableAssignment+
//;

VariableAssignment returns Variable: //Serves as a basis to retain results and to be the basis for lines in the dsl
	{VariableAssignment} 'var' name=ID '=' exp=Exp
;

Exp returns Expression: //Addition and subtraction - Can boil down to MultDiv
	MultDiv (('+' {Plus.left=current}| '-' {Minus.left=current}) right=MultDiv)*
;

MultDiv returns Expression: //Multiplication and devision - Can boil down to MultDiv
	Primary (('*' {Multiplication.left=current}| '/' {Division.left=current}) right=Primary)*
;

Primary returns Expression: //Numbers and things that should be computed down to numbers before use
	Number | Parenthesis | VariableUse | LocalAssignment | ExternalUse
;

Parenthesis returns Expression: //Serves to support the use of parentheses as
```

```
a base
    {Parenthesis} '(' exp=Exp ')'
;

Number returns Expression: //A basic number
    {ExplicitNumber} value=INT
;

VariableUse: //Using a previously defined variable
    {VarUse} ref=[Variable]
;

Assignment returns Variable:
    {Assignment} name=ID '=' exp=Exp
;

LocalAssignment: //This is kind of like a using statement, where an alias is
made for an expression or simmilar that only exists in the body of the let
statement
    {Local} 'let' assignment=Assignment 'in' exp=Exp 'end'
;


```

```
 * generated by Xtext 2.25.0
package dk.sdu.mmmi.mdsd.generator

import dk.sdu.mmmi.mdsd.math.Division

/**
 * Generates code from your model files on save.
 *
 * See https://www.eclipse.org/Xtext/documentation/
303_runtime_concepts.html#code-generation
 */
class MathGenerator extends AbstractGenerator {

    static Map<String, Integer> variables = new HashMap();

    override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext context) {
        val program = resource.allContents.filter(Program).next
        val className = program.programName.name
        // Append Header to class
        val math_expressionClass = getMathClass(program, className)
        fsa.generateFile(className + ".java", math_expressionClass)
    }

    def getMathClass(Program program, String className) {
        val contents = '''
            «getHeader()»
            package math_expression;
            public class «className»{
                «getVariableDeclarations(program.variableAssignments)»

                «IF program.externals.length() > 0»
                    private External external;

                    public «className»(External external){
                        this.external = external;
                    }

                    «getExternalInterface(program)»
                «ENDIF»

                public void compute(){
                    «FOR instantiation : getVariableInstantiations(program)»
                        «instantiation»;
                    «ENDFOR»
                }

            }
        '''
        //println(contents)
        return contents
    }

    def getVariableInstantiations(Program program){
        var vars = program.variableAssignments
        return vars.compute()
    }

    def getExternalInterface(Program program) {
        return '''
            public interface External{
                «FOR external : program.externals»
                    public int «getExternalSignature(external)»;
                «ENDFOR»
            }
        '''
    }

    def getExternalSignature(External external) {
        var returnValue = external.name + "("
        if (external.parameters.length == 1) {
            returnValue += "int n"
        } else if (external.parameters.length == 2) {

            returnValue += "int n, int m"
        }

        returnValue += ")"
        return returnValue
    }

    def getHeader() {
        return '''
            /*
             * Generated by xtend by the generator made by 'Frederik Alexander Hounsvad' 'frhou18@student.sdu.dk'
             * All Rights reserved
            */
        '''
    }

    def getVariableDeclarations(EList<Variable> list) {
        return '''
            «FOR variable : list»
                public int «variable.name»;
            «ENDFOR»
        '''
    }

    def List<String> compute(EList<Variable> variables) {
        var values = new ArrayList<String>();

        for (varass : variables) {
            values.add(varass.name + " = " + ComputeExp(varass))
        }
        return values
    }

    // Plus
    def static dispatch String ComputeExp(Plus exp) {
        return '''(«exp.left.ComputeExp()» + «exp.right.ComputeExp()»)'''
    }

    // Minus
    def static dispatch String ComputeExp(Minus exp) {
        return '''(«exp.left.ComputeExp()» - «exp.right.ComputeExp()»)'''
    }

    // Multiplication
    def static dispatch String ComputeExp(Multiplication exp) {
        return '''(«exp.left.ComputeExp()» * «exp.right.ComputeExp()»)'''
    }

    // Division
    def static dispatch String ComputeExp(Division exp) {
        return '''(«exp.left.ComputeExp()» / «exp.right.ComputeExp()»)'''
    }

    // ExplicitNumber
    def static dispatch String ComputeExp(ExplicitNumber exp) {
        return '''«exp.value»'''
    }

    // Parenthesis
    def static dispatch String ComputeExp(Parenthesis exp) {
        return '''(«exp.getExp.ComputeExp()»)'''
    }

    // VarUse
    def static dispatch String ComputeExp(VarUse exp) {
        return '''(«exp.ref.ComputeExp()»)'''
    }

    // Let
    def static dispatch String ComputeExp(Local exp) { // Let
        return '''(«exp.exp.ComputeExp()»)'''
    }

    // Variable
    def static dispatch String ComputeExp(Variable exp) {
        return '''(«exp.exp.ComputeExp()»)'''
    }

    def static dispatch String ComputeExp(ExternalUse exp) {
        var sb = new StringBuilder()
        sb.append("(external.").append(exp.ref.name).append("(")
        switch exp.ref.parameters.length(){
            case 1: sb.append(ComputeExp(exp.exp.get(0)))
            case 2: sb.append(ComputeExp(exp.exp.get(0))).append(",").append(ComputeExp(exp.exp.get(1)))
        }
        sb.append("))")
        return sb.toString()
    }
}
```

```
 * generated by Xtext 2.26.0
package dk.sdu.mmmi.mdsd.scoping

import dk.sdu.mmmi.mdsd.math.Assignment

/**
 * This class contains custom scoping description.
 *
 * See https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#scoping
 * on how and when to use it.
 */
class MathScopeProvider extends AbstractMathScopeProvider {

    override IScope getScope(EObject context, EReference reference){
        var scope = super.getScope(context, reference)
        if(context instanceof VarUse){

            var IScope returnScope = null

            var letDefinition = EcoreUtil2.getContainerOfType(context, Local)
            var letVariable = EcoreUtil2.getContainerOfType(context, Assignment)
            if(letDefinition !== null && letVariable !== letDefinition.assignment){
                returnScope = addLetDefinition(letDefinition, context)
            }else{
                if(letDefinition !== null){
                    letDefinition = EcoreUtil2.getContainerOfType(letDefinition.eContainer, Local)
                }
                if(letDefinition !== null){
                    returnScope = addLetDefinition(letDefinition, context)
                }else{
                    returnScope = getVariableAssignmentsInScope(context);
                }
            }
            return returnScope
        }
        return scope;
    }
    protected def IScope addLetDefinition(Local letDefinition, EObject context){
        val containingLet = EcoreUtil2.getContainerOfType(letDefinition.eContainer, Local)

        if(containingLet === null){
            return Scopes.scopeFor(#[letDefinition.assignment],
getVariableAssignmentsInScope(context))
        }else{
            return Scopes.scopeFor(#[letDefinition.assignment],
addLetDefinition(containingLet, context))
        }
    }

    protected def IScope getVariableAssignmentsInScope(EObject context){
        val root = EcoreUtil2.getRootContainer(context);
        val List<EObject> candidates = new ArrayList();
        //Get all variableAssignments
        for(VariableAssignment assignment:
EcoreUtil2.getAllContentsOfType(root, VariableAssignment)){
            candidates.add(assignment as EObject)
        }

        //Should generate a list of all variables that are not let (ie the
var difinitions that are global)
        val List<EObject> variableAssignments = candidates
        .filter(variable | variable !== EcoreUtil2.getContainerOfType(context,
Variable))
        .toList()

        return Scopes.scopeFor(variableAssignments)
    }
}

```

```
 * generated by Xtext 2.26.0
package dk.sdu.mmmi.mdsd.validation

import dk.sdu.mmmi.mdsd.math.*

/**
 * This class contains custom validation rules.
 *
 * See https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#validation
 */
class MathValidator extends AbstractMathValidator {

//	public static val INVALID_NAME = 'invalidName'
//
//	@Check
//	def checkGreetingStartsWithCapital(Greeting greeting) {
//		if (!Character.isUpperCase(greeting.name.charAt(0))) {
//			warning('Name should start with a capital',
//					MathPackage.Literals.GREETING__NAME,
//					INVALID_NAME)
//		}
//	}

	public static val DUPLICATE_NAME = 'duplicateName'

	@Check
	def GlobalVarDuplicate(VariableAssignment varAss){
		var base = EcoreUtil2.getContainerOfType(varAss, Program)
		if(base.variableAssignments.filter[it !== varAss && it.name == varAss.name ].toList.size > 0){
			println("Should have err")
			error('Global variables cannot be assigned with the same name', MathPackage.Literals.VARIABLE__NAME, DUPLICATE_NAME)
		}
	}

	public static val WRONG_PARAMETER_COUNT = 'wrongParamaterCount'

	@Check
	def ValidateExternalParameterCount(ExternalUse use){
		var parent = use.ref
		if (parent.parameters.length() !== use.exp.length()){
			error('Call to external function with incorrect number of parameters', MathPackage.Literals.EXTERNAL_USE__EXP, WRONG_PARAMETER_COUNT)
		}
	}

}

```