Assigment 03 — Nikolai Emil Damm

My code

GitHub — devantler/mdsd-assignment03

Assignment status

I have solved the assignment such that all tests pass by utilizing my own version of Assignment 2 as the base.

OBS! I could not make Externals work without adding parenthesis to the DSL code for External parameters in `Test31.math` and `Test34.math`. I know this happens because my `ValueExpression` does not allow expressions without parameters, and adding that to my grammar rules gave me left-hand recursion errors.

My grammar language

```
Model:
    'program' name=ID
    externalDefinitions+=ExternalDefinition*
    variables+=GlobalVariable*;

GlobalVariable returns Variable:
    {GlobalVariable} 'var' name=ID '='
expression=AdditionAndSubtractionExpression;

LocalVariable returns Variable:
    {LocalVariable} 'let' name=ID '='
local_expression=AdditionAndSubtractionExpression 'in'
    expression=AdditionAndSubtractionExpression 'end';

AdditionAndSubtractionExpression returns Expression:
    MultiplicationAndDivisionExpression (({Plus.left=current} '+' |
{Minus.left=current} '-')
    right=MultiplicationAndDivisionExpression)*;

MultiplicationAndDivisionExpression returns Expression:
    ValueExpression (({Multiplication.left=current} '*' |
{Division.left=current} '/') right=ValueExpression)*;

ValueExpression returns Expression:
    {Parenthesis} '('
parenthesizedExpression=AdditionAndSubtractionExpression ')' | {Number}
value=INT | LocalVariable
    | VariableReference | External;

VariableReference:
    variable=[Variable];
```

```
ExternalDefinition:
    'external' (PiExternalDefinition | SqrtExternalDefinition |
PowExternalDefinition)
;

PiExternalDefinition returns ExternalDefinition:
    {PiExternalDefinition} name='pi''()'
;

SqrtExternalDefinition  returns ExternalDefinition:
    {SqrtExternalDefinition} name='sqrt''('param1='int'')'
;

PowExternalDefinition returns ExternalDefinition:
    {PowExternalDefinition} name='pow''('param1='int','param2='int'')'
;

External:
    PiExternal | SqrtExternal | PowExternal
;

PiExternal returns External:
    {PiExternal} name='pi''()'
;

SqrtExternal returns External:
    {SqrtExternal} name='sqrt''('param1=ValueExpression')'
;

PowExternal returns External:
    {PowExternal}
name='pow''('param1=ValueExpression','param2=ValueExpression')'
;
```

My generator

```
class MathGenerator extends AbstractGenerator {

    static Map<String, String> variables = new HashMap;

    override doGenerate(Resource input, IFileSystemAccess2 fsa,
IGeneratorContext context) {
        val model = input.allContents.filter(Model).next
        fsa.generateFile("math_expression/" + model.name + ".java",
model.compile())
    }

    def static String compile(Model model) '''
        package math_expression;

        public class «model.name» {
```

```
            «FOR globalVariable : model.variables.filter[it instanceof
GlobalVariable]»
                public int «globalVariable.name»;

            «ENDFOR»
            «IF !model.externalDefinitions.isNullOrEmpty»
                private External external;

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

                public interface External {
                    «FOR externalDefinition:model.externalDefinitions»
                        «generateExternalDefinition(externalDefinition)»
                    «ENDFOR»
                }

        «ENDIF»
            public void compute() {
            «FOR variable : model.variables»
            «"  "»«variable.name» = «(variable as
GlobalVariable).compile»;
            «ENDFOR»
            }
        }
    '''

    protected def static CharSequence
generateExternalDefinition(ExternalDefinition externalDefinition) {
        val parameters = switch externalDefinition {
            SqrtExternalDefinition: externalDefinition.param1 + " n"
            PowExternalDefinition: externalDefinition.param1 + " n1, " +
externalDefinition.param2 + " n2"
            default: ""
        }
        '''public int «externalDefinition.name»(«parameters»);'''
    }

    def static String compile(GlobalVariable globalVariable) {
        val expressionValue = globalVariable.expression.compileExp(new
HashMap)
        variables.put(globalVariable.name, expressionValue)
        return variables.get(globalVariable.name)
    }

    def static dispatch String compileExp(Expression expression,
Map<String, String> localVariables) {
        switch expression {
            Plus:
                expression.left.compileExp(localVariables) + ' + ' +
expression.right.compileExp(localVariables)
            Minus:
```

```
                expression.left.compileExp(localVariables) + ' - ' +
expression.right.compileExp(localVariables)
            Multiplication:
                expression.left.compileExp(localVariables) + ' * ' +
expression.right.compileExp(localVariables)
            Division:
                expression.left.compileExp(localVariables) + ' / ' +
expression.right.compileExp(localVariables)
            Number:
                expression.value.toString
            Parenthesis:
                '(' +
expression.parenthesizedExpression.compileExp(localVariables) + ')'
        }
    }

    def dispatch static String compileExp(External external, Map<String,
String> localVariable){
        var result = 'this.external.'
        switch external {
            PiExternal: return result + external.name + '()'
            SqrtExternal: return result + external.name + '(' +
external.param1.compileExp(localVariable) + ')'
            PowExternal: return result + external.name + '(' +
external.param1.compileExp(localVariable) + ', ' +
external.param2.compileExp(localVariable) + ')'
        }
    }

    def dispatch static String compileExp(Variable variable, Map<String,
String> localVariables) {
        val nestedVariables = new HashMap(localVariables);
        if (variable instanceof LocalVariable) {
            nestedVariables.put(variable.name,
variable.local_expression.compileExp(nestedVariables))
        }
        variable.expression.compileExp(nestedVariables)
    }

    def dispatch static String compileExp(VariableReference reference,
Map<String, String> localVariables) {
        val globalVariable = variables.get(reference.variable.name)
        val localVariable = localVariables.get(reference.variable.name)
        switch reference.variable {
            LocalVariable:
                localVariable !== null ? '(' + localVariable + ')' : '(' +
globalVariable + ')'
            GlobalVariable:
                globalVariable !== null
                    ? '(' + globalVariable + ')'
                    : '(' + reference.variable.compileExp(localVariables)
+ ')'
        }
```

```
    }
}
```

My Scope Provider

```
class MathScopeProvider extends AbstractMathScopeProvider {

    override IScope getScope(EObject context, EReference reference) {
        switch (reference) {
            case Literals.VARIABLE_REFERENCE__VARIABLE:
context.getVariableScope(true)
            default: super.getScope(context, reference)
        }
    }

    def IScope getVariableScope(EObject object, boolean first) {
        val nextVariable = first ? EcoreUtil2.getContainerOfType(object,
Variable) : EcoreUtil2.getContainerOfType(object.eContainer, Variable);
        if (nextVariable instanceof LocalVariable) {
            return Scopes.scopeFor(#[nextVariable],
nextVariable.getVariableScope(false));
        } else {
            return (nextVariable as
GlobalVariable).getGlobalVariableScope;
        }
    }

    def IScope getGlobalVariableScope(GlobalVariable globalVariable) {
        val model = EcoreUtil2.getRootContainer(globalVariable) as Model;
        val globalVariables = model.variables.filter[it.name !==
globalVariable.name].toList;
        return Scopes.scopeFor(globalVariables)
    }
}
```

My Validator

```
class MathValidator extends AbstractMathValidator {
    @Check
    def noRepeatedGlobalVariablwes(GlobalVariable globalVariable) {
        val model = EcoreUtil2.getRootContainer(globalVariable) as Model;
        val globalVariables = model.variables;
        var occurences = 0;
        for (gv : globalVariables) {
            if(gv.name == globalVariable.name){
                occurences++;
            }
        }
        if(occurences > 1){
```

```
                error("Multiple global variables cannot share the same name.",
globalVariable, Literals.VARIABLE__NAME)
            }
        }
    }
```