Assignment 3: Math Compiler

This assignment was about making a compiler for Ulrik's math interpreter solution as a starting point.

The individual tasks are listed below:

1. Extend the grammer to have multiple mathematical expressions. Each with a textual label.
2. Make a code generator that generates valid Java-class that contains a public method 'calculate' that computes the value of each mathematical expression and prints it to the console along with the corresponding label.
3. Optionally include support for external functions
4. Optionally support the functional-style variable assignment

Status of Assignment

The math compiler is able to handle all of the aforementioned tasks. That is; (1) multiple mathematical expressions, (2) generate the Java-class, (3) supports external functions and (4) supports functional-style variable assignments. The functional-style variable assignments was solved by using the Function-interface from Java to deal with scoping.

Xtext File

Click to see xtext-file on github

The xtext file was extended to include 'Declaration', 'ExternalDef', 'ExternalUse', 'Type', 'Parameter', 'ResultStatement'.

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

generate mathAssignmentLanguage
"http://www.sdu.dk/mmmi/mdsd/MathAssignmentLanguage"

MathExp:
    declarations+=Declaration*
;

Declaration:
    Type | ExternalDef | ResultStatement
;

ExternalDef:
    'external'name=ID '(' parameters+=Parameter (','
parameters+=Parameter)* ')'
;

Parameter:
    type=[Type] parameterName=ID
;
```

```
Type:
    'type' name=ID
;

ResultStatement returns ResultStatement:
    'result' label=STRING 'is' exp=Exp
;

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

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

Primary returns Expression:
    Number | Parenthesis | VariableBinding | VariableUse | ExternalUse
;

VariableUse returns Expression:
    {Var} id=ID
;

VariableBinding returns Expression:
    {Let} 'let' id=ID '=' binding=Exp 'in' body=Exp 'end'
;

ExternalUse returns Expression:
    {ExternalUse} external=[ExternalDef] '(' arguments+=Exp (','
arguments+=Exp)* ')'
;

Parenthesis returns Expression:
    '(' Exp ')'
;

Number returns Expression:
    {Num} value=INT
;
```

Xtend Generator File

Click to see xtend-generator file on github

```
/*
 * generated by Xtext 2.21.0
 */
```

```
package dk.sdu.mmmi.mdsd.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext
import dk.sdu.mmmi.mdsd.mathAssignmentLanguage.MathExp
import dk.sdu.mmmi.mdsd.mathAssignmentLanguage.Expression
import dk.sdu.mmmi.mdsd.mathAssignmentLanguage.Plus
import dk.sdu.mmmi.mdsd.mathAssignmentLanguage.Minus
import dk.sdu.mmmi.mdsd.mathAssignmentLanguage.Mult
import dk.sdu.mmmi.mdsd.mathAssignmentLanguage.Div
import dk.sdu.mmmi.mdsd.mathAssignmentLanguage.Num
import dk.sdu.mmmi.mdsd.mathAssignmentLanguage.Var
import dk.sdu.mmmi.mdsd.mathAssignmentLanguage.Let
import dk.sdu.mmmi.mdsd.mathAssignmentLanguage.ResultStatement
import java.util.HashSet
import dk.sdu.mmmi.mdsd.mathAssignmentLanguage.ExternalDef
import dk.sdu.mmmi.mdsd.mathAssignmentLanguage.ExternalUse

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

    override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {

        val math = resource.allContents.filter(MathExp).next // Root level of metamodel instance

        fsa.generateFile("MathComputation.java", math.compile)

        fsa.generateFile("compileAndRun.sh",
            '''
            #!/bin/bash
            javac MathComputation.java
            java MathComputation
            '''
        )
    }

    def compile(MathExp math) {
        val className = "MathComputation" // Name of the generated class
        val externalDefs = math.declarations.filter(ExternalDef)
        val resultStatements = math.declarations.filter(ResultStatement)

        '''
        /*
        * AUTO-GENERATED CODE!
        */

        /*
        * Imports
        */
        import java.util.function.Function;

        /*
        * Class
        */
        public class «className» {

            /*
            * Fields
            */
            «IF externalDefs.length > 0»
            private Externals externals;
            «ENDIF»

            /*
            * Constructors
            */
            «IF externalDefs.length > 0»
            public «className»(Externals externals) {
                this.externals = externals;
            }

            /*
            * External functions
            */
            public static interface Externals {
                «FOR externalDef : externalDefs»
                «generateExternalSignature(externalDef)»;
                «ENDFOR»
            }
            «ELSE»
            public «className»() { }
            «ENDIF»


            /*
            * Public methods
            */
            public void compute() {
                // Call compute on each result-statement
                «FOR resultStatement : resultStatements»
                System.out.println("«resultStatement.label» " +
compute«resultStatement.label.convertTolegalJavaIdentifier.toFirstUpper»())
;
                «ENDFOR»
            }

            /*
            * Result statements
            */
            «FOR resultStatement : resultStatements»
                «generatePrivateMethod(resultStatement)»
            «ENDFOR»

            /*
            * Main methods
            */
            «IF externalDefs.length > 0»
            public static void main(String[] args) {
                new «className»(new Externals() {
                    @Override
                    «FOR externalDef : externalDefs»
                    «generateExternalSignature(externalDef)» {
                        // TODO: Implement method
                        throw new UnsupportedOperationException();
                    }
                    «ENDFOR»
                }).compute();
            }
            «ELSE»
            public static void main(String[] args) {
                new «className»().compute();
            }
            «ENDIF»
        }
        '''
    }

    def String generateExternalSignature(ExternalDef exDef) {
        val parameters = exDef.parameters

        val parameterString = new StringBuilder()
        for (parameter : parameters) {
            if (parameterString.length > 0) parameterString.append(", ")
            parameterString.append(parameter.type.name).append("
").append(parameter.parameterName)
        }

        return
        '''public int «exDef.name»(«parameterString»)'''
    }

    def String generatePrivateMethod(ResultStatement r) {
        '''
        private int
compute«r.label.convertTolegalJavaIdentifier.toFirstUpper»() {
            return «r.exp.compile»;
        }
        '''
    }

    def String convertTolegalJavaIdentifier(String s) {
        val validChars = "[a-z]|[A-Z]|\\d|[_]"
        val illegalChars = new HashSet()

        var validIdentifier = new String(s)

        // Find illegal chars
        for (var i = 0; i < s.length(); i++) {
            val myChar = s.substring(i, i+1);
            if (!myChar.matches(validChars)) {
                illegalChars.add(myChar);
            }
        }

        // Remove illegal chars
        for (String illegalChar : illegalChars) {
            validIdentifier = validIdentifier.replace(illegalChar, "");
        }

        return validIdentifier
    }

    def String compile(Expression exp) {
        "(" + switch (exp) {
            Plus: '''«exp.left.compile»+«exp.right.compile»'''
            Minus: '''«exp.left.compile»-«exp.right.compile»'''
            Mult: '''«exp.left.compile»*«exp.right.compile»'''
            Div: '''«exp.left.compile»/«exp.right.compile»'''
            Num: '''«exp.value»'''
            Var: '''t''' // t is the variable used by the generated java-code for let expressions below
            Let: {
                '''
                new Function<Integer, Integer>() {
                    @Override
                    public Integer apply(Integer t) {
                        return «exp.body.compile»;
                    }
                }.apply(«exp.binding.compile»)'''
            }
            ExternalUse: {
                val extArguments = new StringBuilder()
                for (extExp : exp.arguments) {
                    if (extArguments.length > 0) extArguments.append(", ")
                    extArguments.append("(").append(extExp.compile).append(")")
                }
                '''externals.«exp.external.name»(«extArguments»)'''
             }

            default: throw new Error("Compile: Invalid expression")
        } + ")"
    }
}
```

Example of a Generated Java Program

A sample DSL program is presented first, then the corresponding generated Java code is presented second.

Click to see DSL-program file on github

```
type int
external power(int a, int b)
external sqrt(int a)

result "Basic arithmetics 1" is 1 + (6 / 3) * 5 - 2

result "Basic arithmetics 2" is 210 * 10 + 35 + (5 - (2 - 1))

result "External functions" is 1 + power(2, 3) * sqrt(9)

// 14
result "Functional style 1" is
let myVar = 1+2 in
    2 * let x = myVar*2 in
        x + 1
    end
end

// 21
result "Functional style 2" is
let myVar = 1+2 in
    3 * let x = myVar*2 in
        x + 1
    end
end

// 13
result "Functional style 3" is
let x = 2 * 5 + 4 / 2 in x + 1 end

// 9
result "Functional style 4" is
let myVar = 2+2 in
    let x = myVar*2 in
        x + 1
    end
end

// 11
result "Functional style 5" is
let myVar = 3+2 in
    let x = myVar*2 in
        x + 1
    end
end
* 1
```

```
// 10
result "Functional style nested variable scope" is
let x = 1+2 in
    let x = x*x in
        x + 1
    end
end

// 10
result "Functional style in-line" is 1 + (6 / 3) * 5 - let x = 1 in x end

// 202
result "Functional style + external functions" is
let myVar = 3+2 * sqrt(9) in // 9
    let x = myVar*2 + power(myVar, 2) in // 18 + 81
        x + 1 * sqrt(4) // 99 + 2
    end
end
* 2
```

The generated Java-file is presented in the next snippet. Click to see generated Java-file on github

```
/*
* AUTO-GENERATED CODE!
*/

/*
* Imports
*/
import java.util.function.Function;

/*
* Class
*/
public class MathComputation {

    /*
    * Fields
    */
    private Externals externals;

    /*
    * Constructors
    */
    public MathComputation(Externals externals) {
        this.externals = externals;
    }

    /*
    * External functions
    */
    public static interface Externals {
        public int power(int a, int b);
        public int sqrt(int a);
    }


    /*
    * Public methods
    */
    public void compute() {
        // Call compute on each result-statement
        System.out.println("Basic arithmetics 1 " +
computeBasicarithmetics1());
        System.out.println("Basic arithmetics 2 " +
computeBasicarithmetics2());
        System.out.println("External functions " +
computeExternalfunctions());
        System.out.println("Functional style 1 " +
computeFunctionalstyle1());
        System.out.println("Functional style 2 " +
computeFunctionalstyle2());
        System.out.println("Functional style 3 " +
computeFunctionalstyle3());
        System.out.println("Functional style 4 " +
computeFunctionalstyle4());
        System.out.println("Functional style 5 " +
computeFunctionalstyle5());
        System.out.println("Functional style nested variable scope " +
computeFunctionalstylenestedvariablescope());
        System.out.println("Functional style in-line " +
computeFunctionalstyleinline());
        System.out.println("Functional style + external functions " +
computeFunctionalstyleexternalfunctions());
    }

    /*
    * Result statements
    */
    private int computeBasicarithmetics1() {
        return (((1)+(((6)/(3))*(5)))-(2));
    }
    private int computeBasicarithmetics2() {
        return ((((210)*(10))+(35))+((5)-((2)-(1))));
    }
    private int computeExternalfunctions() {
        return ((1)+((externals.power(((2)), ((3))))*
(externals.sqrt(((9))))));
    }
    private int computeFunctionalstyle1() {
        return (new Function<Integer, Integer>() {
            @Override
            public Integer apply(Integer myVar) {
                return ((2)*(new Function<Integer, Integer>() {
                    @Override
                    public Integer apply(Integer x) {
                        return ((x)+(1));
                    }
                }.apply(((myVar)*(2)))));
            }
        }.apply(((1)+(2))));
    }
    private int computeFunctionalstyle2() {
        return (new Function<Integer, Integer>() {
            @Override
            public Integer apply(Integer myVar) {
                return ((3)*(new Function<Integer, Integer>() {
                    @Override
                    public Integer apply(Integer x) {
                        return ((x)+(1));
                    }
                }.apply(((myVar)*(2)))));
            }
        }.apply(((1)+(2))));
    }
    private int computeFunctionalstyle3() {
        return (new Function<Integer, Integer>() {
            @Override
            public Integer apply(Integer x) {
                return ((x)+(1));
            }
        }.apply((((2)*(5))+((4)/(2)))));
    }
    private int computeFunctionalstyle4() {
        return (new Function<Integer, Integer>() {
            @Override
            public Integer apply(Integer myVar) {
                return (new Function<Integer, Integer>() {
                    @Override
                    public Integer apply(Integer x) {
                        return ((x)+(1));
                    }
                }.apply(((myVar)*(2))));
            }
        }.apply(((2)+(2))));
    }
    private int computeFunctionalstyle5() {
        return ((new Function<Integer, Integer>() {
            @Override
            public Integer apply(Integer myVar) {
                return (new Function<Integer, Integer>() {
                    @Override
                    public Integer apply(Integer x) {
                        return ((x)+(1));
                    }
                }.apply(((myVar)*(2))));
            }
        }.apply(((3)+(2))))*(1));
    }
    private int computeFunctionalstylenestedvariablescope() {
        return (new Function<Integer, Integer>() {
            @Override
            public Integer apply(Integer x) {
                return (new Function<Integer, Integer>() {
                    @Override
                    public Integer apply(Integer x) {
                        return ((x)+(1));
                    }
                }.apply(((x)*(x))));
            }
        }.apply(((1)+(2))));
    }
    private int computeFunctionalstyleinline() {
        return (((1)+(((6)/(3))*(5)))-(new Function<Integer, Integer>() {
            @Override
            public Integer apply(Integer x) {
                return (x);
            }
        }.apply((1))));
    }
    private int computeFunctionalstyleexternalfunctions() {
        return ((new Function<Integer, Integer>() {
            @Override
            public Integer apply(Integer myVar) {
                return (new Function<Integer, Integer>() {
                    @Override
                    public Integer apply(Integer x) {
                        return ((x)+((1)*(externals.sqrt(((4))))));
                    }
                }.apply((((myVar)*(2))+(externals.power(((myVar)),
((2)))))));
            }
        }.apply(((3)+((2)*(externals.sqrt(((9))))))))*(2));
    }

    /*
    * Main methods
    */
    public static void main(String[] args) {
        new MathComputation(new Externals() {
            @Override
            public int power(int a, int b) {
                // TODO: Implement method
                throw new UnsupportedOperationException();
            }
            public int sqrt(int a) {
                // TODO: Implement method
                throw new UnsupportedOperationException();
            }
        }).compute();
    }
}
```