# MDSD Takehome Report

Marcus Lomstein Jensen, marcj18@student.sdu.dk

## Xtext Grammar:

### (a) Does your grammar support all of the example programs? If not what are the limitations?

All example programs generate completely, all tests pass with the exception of one. I believe the one test that fails is a mistake, due to a missing input. The Story seems to expect 3 inputs, but only 2 is given by the test, resulting in what I assume is some sort of end-of-stream error. Adding a third input causes the test to pass.

I am confident the DSL is powerful and robust enough, though I haven't done much testing beyond running the unit tests and poking around every now and then.

### (b) How did you implement operator precedence and associativity?

Precedence and associativity is implemented using code from my semester project, which in turn was taken from the assignments. First of all, precedence is implemented by essentially implementing a set of cascading rules, described by the documentation as a delegation pattern, where each rule with higher precedence refers to a set of rules below it with lower precedence. Associativity is, as far as I understand, implemented through the use of left-factoring, but truth be told I don't truly understand how it works.

### (c) How did you implement the syntax of variables (ID in rule Exp of the IF22 BNF), such that they can refer both to parameters of scenarios and local variables?

Variables are implemented as two different types of bindings. VarBindings and Parameters. VarBindings are bindings of variables to a particular scenario. Parameters are bindings of scenario input parameters, received through the constructor of the scenario, and both of these can be referenced in expression using cross-references. Both type of references are defined at the top of the scenario class, and populated as the program runs. To differentiate between the many bindings, I make use of prefixes such that it is possible to tell the difference between question inputs, local variables, and parameters.

### (d) How did you implement the syntax of two statements, such that they refer to announcement/question/end/scenario? (first ID in rule Target of the IF22 BNF)

The "to" statements are implemented by defining a class of "Targetables", which can be either a Scenario and a Location. A Location can then be either an Announce, a Question, or an End.

## Scoping Rules

### (a) Did you implement scoping rules that allow variables to refer to variable definition and scenario parameters? If not, what are the limitations?

No, as I am not sure how it would even function.

### (b) Did you implement scoping rules that allow announcement and question statements to call scenarios and reference the called scenario end statements? If not, what are the limitations?

Yes, I implemented very rudimentary scoping rules, that allows for the cross-referencing of scenarios and locations within said scenarios. The implementation is fairly basic, making use of simple if-statements, but it does what it needs to do.

### (c) Describe your implementation of any scoping rules included with your system.

There are effectively two scoping rules implemented. One of the rules are that when we're in the context of a Target rule, and working with the Target attribute of the Target rule, we scope for all of the scenarios, with the default scope as the outer scope in order to still allow access to the locations within the current scenario. The second rule defines that within the context of an EndingTarget rule and the From attribute of said rule, we scope for all the locations within the scenario that the parent Target rule references.

## Type Inference

### (a) What validation rule did you implement, if any?

I have implemented all but a few of the expression validation rules. Specifically, I have not implemented the rules for validating that the 'this' statement as well as variables behave consistently. Furthermore, I have not implemented any rules for validating expressions, and as such it is possible to write code in the DSL that will generate illegal java code. The one that ensures that Type keywords are only used within the context of input validations is particularly interesting, as it effectively works by checking if the keyword is a child of a Question, and then checking if the Questions validate attribute has the keyword as one of it's children, which is hilariously hacky. An alternative solution would have required changing the grammar, which I am hesitant to do at this point (06:34 AM).

### (b) Did you implement validation for the correct use of keywords such as this and Type keywords (number, text)? If not, what are the limitations? If yes, briefly describe your approach

I have implemented validation rules that ensure that 'this' statements are only used within the context of a Target rule, as well as Type keywords within the context of input validation. It works by effectively traversing the tree down from key points, in order to ensure that the point has a maximum amount of a particular type of children. For instance, checking that there is only one

Type keyword is done like this. Checking ‘this’ is done simply by checking if it has a Target parent somewhere up the hierarchy.

(c) Did you implement validation with type inferencing? If yes, does it correctly check the types for all expressions (including input validation and conditions) in the provided IF22 examples? If not, what are the problems?

Yes. It is implemented by comparing the output type of the expression, with the parameter or variable type of the scenario, question, or function.

(d) If you implemented validation with type inferencing, briefly describe your approach

To check that the parameters match with the arguments, I first check to ensure that the amount of parameters and arguments are the same. If this passes, I loop over them all and compare them individually as described above. Any that does not match correctly are marked as erroneous.

## Generator

(a) Does your code generator correctly generate code for all of the examples provided, and are you confident that it will also work for “similar” programs? If not, then what limitations are there?

Yes. I see no reason for why it wouldn’t work for similar programs, except for the issues described with expression validation.

(b) Briefly describe how your code generator works.

The generator works by first generating boilerplate classes, such as the Scenario and Game classes heavily based on the material provided for the exam, as well as the External interface. Once this is done, classes inheriting from the Scenario class are generated for each of the scenarios within the story. The first scenario is generated as the main one, and this is the one that the Game class begins with. The scenario classes boilerplate is generated fairly trivially, with the most notable method being used is traversing the tree of the entire scenario, in order to find and define all bindings at the top of the class. Once the trivial boilerplate is set up, the main switch case is defined inside a try-catch block. The switch case is generated by generating individual cases as three different types of “Locations”; either Announce, Question, or End, each generated with a dedicated dispatch function. These three functions then further all cover printing a message of present. The Announce and Question methods additionally further branch out into Target generation, and finally the Question method branches out into the reading and parsing of input.

6. Implementation: include your Xtext grammar file and all implemented Xtend files (scoping, type validation, generator, and any additional file).

Grammar

```
grammar lomzt.mdsd.exam.IF22 with org.eclipse.xtext.common.Terminals

generate iF22 "http://www.mdsd.lomzt/exam/IF22"

Story:
	'story' name=ID
	imports += ExternalImport*
	scenarios += Scenario*
;

ExternalImport:
	'function' name=ID '(' params += Type? (',' params += Type)* ')' ':'
returnType = Type
;

Type:
	{Number} 'number' | {Boolean} 'boolean' | {Text} 'text'
;

Scenario:
	'scenario' name=ID ('(' params += Parameter (',' params +=
Parameter)* ')')? '{'
	bindings += VarBinding*
	locations += Location+
	'}'
;

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

Location:
	Announce | Question | End
;

Announce:
	'announce' name=ID (text=Exp)?  targets += Target+
;
```

```
Question:
      'question' name=ID (text=Exp)? 'as' validate=Exp ('in'
binding=[VarBinding])? targets += Target+
;

End:
      'end' name=ID (text=Exp)?
;

Target:
      'to' target=[Targetable] ('('( args+=Exp (',' args+=Exp)*)? ')')?
('if' condition=Exp)? ('{' endingTargets += EndingTarget+ '}')?
;

Targetable:
      Scenario | Location
;

EndingTarget:
      'on' from=[Location] target=Target
;

// These expressions are based on the grammar from the semester project,
with some modifications to implement the concatonation thingies, as well as
some refactoring.
Exp returns Expression:
      EqualsOrNotEquals (({And.left=current} '&&' | {Or.left=current}  '||'
) right=EqualsOrNotEquals)*
;

EqualsOrNotEquals returns Expression:
      Compare (({Equals.left=current} '==' | {NotEquals.left=current}  '!='
) right=Compare)*
;

Compare returns Expression:
      Concatonation (({Greater.left=current} '>' | {Lesser.left=current}
'<' | {GreaterOrEquals.left=current} '>=' | {LesserOrEquals.left=current}
'<=' ) right=Concatonation)*
;

Concatonation returns Expression:
      AdditionSubtraction (({Concat.left=current} '&')
right=AdditionSubtraction)*
;

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

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

Primary returns Expression:
      {TypeValue} type=Type |
      {This} 'this' |
      {BooleanTrue} 'true' |
      {BooleanFalse} 'false' |
      {StringValue} value=STRING |
      {NumberValue} value=INT |
      {VariableUse} ref=[Binding] |
      {Parenthesis} '(' exp=Exp ')' |
      {ExternalCall} func = [ExternalImport] '(' args += Exp? (',' args +=
Exp)* ')' |
      {Not} '!' exp=Exp
;


VarBinding:
      'var' name=ID ':' type=Type
;

Binding:
      VarBinding | Parameter
;
```

Generator

```
/*
 * generated by Xtext 2.26.0
 */
```

```
package lomzt.mdsd.exam.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 lomzt.mdsd.exam.iF22.Story
import lomzt.mdsd.exam.iF22.Scenario
import lomzt.mdsd.exam.iF22.Location
import lomzt.mdsd.exam.iF22.Announce
import lomzt.mdsd.exam.iF22.Question
import lomzt.mdsd.exam.iF22.End
import lomzt.mdsd.exam.iF22.Expression
import lomzt.mdsd.exam.iF22.And
import lomzt.mdsd.exam.iF22.Or
import lomzt.mdsd.exam.iF22.Equals
import lomzt.mdsd.exam.iF22.NotEquals
import lomzt.mdsd.exam.iF22.Greater
import lomzt.mdsd.exam.iF22.Lesser
import lomzt.mdsd.exam.iF22.GreaterOrEquals
import lomzt.mdsd.exam.iF22.LesserOrEquals
import lomzt.mdsd.exam.iF22.Div
import lomzt.mdsd.exam.iF22.Mult
import lomzt.mdsd.exam.iF22.Minus
import lomzt.mdsd.exam.iF22.Plus
import lomzt.mdsd.exam.iF22.Concat
import lomzt.mdsd.exam.iF22.Type
import lomzt.mdsd.exam.iF22.BooleanTrue
import lomzt.mdsd.exam.iF22.BooleanFalse
import lomzt.mdsd.exam.iF22.StringValue
import lomzt.mdsd.exam.iF22.NumberValue
import lomzt.mdsd.exam.iF22.VariableUse
import lomzt.mdsd.exam.iF22.Parenthesis
import lomzt.mdsd.exam.iF22.ExternalCall
import lomzt.mdsd.exam.iF22.Not
import lomzt.mdsd.exam.iF22.Binding
import lomzt.mdsd.exam.iF22.VarBinding
import lomzt.mdsd.exam.iF22.Parameter
import lomzt.mdsd.exam.iF22.Text
import lomzt.mdsd.exam.iF22.This
import java.util.stream.Collectors
import static lomzt.mdsd.exam.Utilities.*
import lomzt.mdsd.exam.iF22.Target
import lomzt.mdsd.exam.iF22.EndingTarget
import java.util.List
import lomzt.mdsd.exam.iF22.TypeValue

/**
 * Generates code from your model files on save.
 *
 * See
https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#code-
generation
 */
class IF22Generator extends AbstractGenerator {

      static final val PACKAGE = "interactive_fiction"
      static final val QUESTION_LAST_INPUT = "__last_input"
      static final val PARAM_LETTERS = "abcdefghijklmnopqrstuvxyzæøå"

      static var paramIndex = 0;
      static Question currentQuestion = null;

      override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext context) {
            val story = resource.allContents.filter(Story).next();

            fsa.generateFile(PACKAGE + "/common/Scenario.java",
scenarioBaseClass())
            fsa.generateFile(PACKAGE + "/" + story.name.toSnakeCase + "/" +
"Game.java", story.compileMainClass)
            fsa.generateFile(PACKAGE + "/" + story.name.toSnakeCase + "/" +
"External.java", story.compileExternalInterface)

            for (scenario : story.scenarios) {
                  fsa.generateFile(PACKAGE + "/" + story.name.toSnakeCase +
"/" + "Scenario" + scenario.name + ".java",
                        scenario.compileScenarioClass)
            }
      }

      static def scenarioBaseClass() '''
            package interactive_fiction.common;

            import java.io.BufferedReader;
            import java.io.InputStreamReader;
            import java.io.IOException;

            public abstract class Scenario {
                  protected static BufferedReader br = new
BufferedReader(new InputStreamReader(System.in));
                  protected String nextInteraction;
                  protected String calledScenarioEnd;

                  public abstract String interact() throws IOException ;

                  public static void changeInput(InputStreamReader
streamReader) {
                        br = new BufferedReader(streamReader);
                  }
            }
      '''

      static def compileMainClass(Story story) '''
            package interactive_fiction.«story.name.toSnakeCase»;

            import java.io.IOException;
            import interactive_fiction.common.*;

            public class Game{
                  public Scenario start;

                  «IF story.hasExternalImports»
                  public Game(External external) {
                        this.start = new
«story.scenarios.get(0).className»(external);
                  }
                  «ELSE»
                  public Game() {
                        this.start = new
«story.scenarios.get(0).className»();
                  }
                  «ENDIF»

                  public void play()  throws IOException {
                        start.interact();
                  }
            }
      '''
```

```
	static def compileScenarioClass(Scenario scenario) '''
		package interactive_fiction.«getParentOfType(scenario,
typeof(Story)).name.toSnakeCase»;

		import java.io.IOException;
		import interactive_fiction.common.*;

		class «scenario.className» extends Scenario {

			External external;
			String «QUESTION_LAST_INPUT»;

			«scenario.compileBindings»

			public «scenario.className»(«String.join(", ",
scenario.params.stream().map(x | x.paramToJavaParam as
CharSequence).collect(Collectors.toList()))») {
				«FOR param : scenario.params»
					«param.bindingName» = «param.name»;
				«ENDFOR»
			}

			public «scenario.className»(«String.join(", ",
scenario.params.stream().map(x | x.paramToJavaParam as
CharSequence).collect(Collectors.toList()))»«addPrefixIfAnyInList('External
external', ', ', scenario.params)») {
				this.external = external;
				«FOR param : scenario.params»
					«param.bindingName» = «param.name»;
				«ENDFOR»
			}

			public String interact() throws IOException {
				String calledScenarioEnd = null;
				nextInteraction =
"«scenario.locations.get(0).name»";
				while(true){
					try {
						switch(nextInteraction){
							«FOR location :
scenario.locations»
								case "«location.name»":
```

```
«location.compileScenarioLocation»
                                          «ENDFOR»
                                    }
                              } catch (Exception ex) {
                                    System.out.println("An error occured,
please try again..");
                                    break;
                              }
                        }

                        return null;
                  }
            }
      '''

      static def hasExternalImports (Story story) {
            return story.imports.size() > 0;
      }

      // i hate this
      static def addPrefixIfAnyInList(String input, String prefix, List
list) {
            if (list.size() > 0) {
                  return prefix + input;
            }else{
                  return input;
            }
      }

      static def compileExternalInterface(Story story) '''

            package interactive_fiction.«story.name.toSnakeCase»;

            public interface External {
                  «FOR importFuncs : story.imports»
                        public «typeToJavaType(importFuncs.returnType)»
«importFuncs.name»(«String.join(", ", importFuncs.params.stream().map(x |
typeToJavaType(x) + " " + getParamLetter()
            ).collect(Collectors.toList()))»);
                  «ENDFOR»
            }
      '''
```

```
	static def getParamLetter() {
		val index = paramIndex++ % PARAM_LETTERS.length()
		return PARAM_LETTERS.charAt(index)
	}

	static def paramToJavaParam(Parameter parameter) {
		return typeToJavaType(parameter.type) + " " + parameter.name
	}

	static def compileBindings(Scenario scenario) {
		val bindings = getChildrenOfType(scenario, typeof(Binding))
		return '''
			«FOR binding : bindings»
				«binding.compileBinding»
			«ENDFOR»
		'''
	}

	static def compileBinding(Binding binding) {
		switch (binding) {
			VarBinding: typeToJavaType(binding.type) + " " +
binding.bindingName + ";"
			Parameter: typeToJavaType(binding.type) + " " +
binding.bindingName + ";"
		}
	}

	static dispatch def compileScenarioLocation(Announce announce) '''
		«IF announce.text !== null»
			«announce.text.compilePrintout»
		«ENDIF»
		«FOR target : announce.targets»
			«target.compileTarget»
		«ENDFOR»
		break;
	'''

	static dispatch def compileScenarioLocation(Question question) '''
		«IF question.text !== null»
			«question.text.compilePrintout»
		«ENDIF»
		«{ currentQuestion = question; ''}»
```

```
            «QUESTION_LAST_INPUT» = br.readLine();
            «question.defineThisBinding» =
«compileConvertInputTo(QUESTION_LAST_INPUT, inputType(question))»;
            «IF !(question.validate instanceof TypeValue)»
            if (!(«question.validate.compileExpression»)) {
                  throw new Exception("Invalid input, please try again.");
            }
            «ENDIF»
            «IF question.binding !== null»
                  «question.binding.bindingName» =
«compileConvertInputTo(QUESTION_LAST_INPUT, inputType(question))»;
            «ENDIF»
            «FOR target : question.targets»
                  «target.compileTarget»
            «ENDFOR»
            break;
      '''

      static def defineThisBinding (Question question) {
            return inputType(question) + " " + question.thisBinding
      }

      static def thisBinding (Question question) {
            '__this_' + question.name
      }

      static dispatch def compileScenarioLocation(End end) '''
            «IF end.text !== null»
                  «end.text.compilePrintout»
            «ENDIF»
            return "«end.name»";
      '''

      static def compilePrintout(Expression exp) '''
            System.out.println(«exp.compileExpression»);
      '''

      static def compileTarget(Target target) '''
            «IF target.condition !== null»
                  if («target.condition.compileExpression») {
                        «target.compileTransition»
                        break;
                  }
```

```
		«ELSE»
			«target.compileTransition»
		«ENDIF»
	'''

	static def compileTransition(Target target) '''
		«IF target.target instanceof Location»
			nextInteraction = "«target.target.name»";
		«ELSEIF target.target instanceof Scenario»
			calledScenarioEnd = new «(target.target as
Scenario).className»(«String.join (",", target.args.stream().map(x |
x.compileExpression).collect(Collectors.toList()))»«addPrefixIfAnyInList('e
xternal', ', ', target.args)»).interact();
			«FOR et : target.endingTargets»
				«et.compileEndingTarget»
			«ENDFOR»
		«ENDIF»
	'''

	static def compileEndingTarget(EndingTarget endingTarget) '''
		if (calledScenarioEnd.equals("«endingTarget.from.name»")) {
			nextInteraction = "«endingTarget.target.target.name»";
			break;
		}
	'''

	static def className(Scenario scenario) {
		return "Scenario" + scenario.name
	}

	static def readBr(Type readAs) {
		switch (readAs) {
			lomzt.mdsd.exam.iF22.Number:
"Integer.parseInt(br.readLine())"
			lomzt.mdsd.exam.iF22.Boolean:
"Boolean.parseBoolean(br.readline())"
			Text: "br.readLine()"
		}
	}

	static def bindingName(Binding binding) {
		switch (binding) {
			VarBinding: "__var_" + binding.name
```

```
				Parameter: "__param_" + binding.name
			}
	}

	static dispatch def compileConvertInputTo(String str, Type type) {
		switch (type) {
			lomzt.mdsd.exam.iF22.Number: 'Integer.parseInt(' + str +
')'
			lomzt.mdsd.exam.iF22.Boolean: 'Boolean.parseBoolean(' +
str + ')'
			Text: str + '.toString()'
		}
	}

	static dispatch def compileConvertInputTo(String str, String
javaType) {
		switch (javaType) {
			case 'int': 'Integer.parseInt(' + str + ')'
			case 'boolean': 'Boolean.parseBoolean(' + str + ')'
			case 'String': str + '.toString()'
		}
	}

	static dispatch def properEquals(Equals eq) {
		val anyString = getExpressionOutputType(eq.left) == 'String' ||
getExpressionOutputType(eq.right) == 'String'
		if (anyString) {
			return
'''«eq.left.compileExpression».equals(«eq.right.compileExpression»)'''
		}else{
			return eq.left.compileExpression + " == " +
eq.right.compileExpression
		}
	}

	static dispatch def properEquals(NotEquals neq) {
		val anyString = getExpressionOutputType(neq.left) == 'String'
|| getExpressionOutputType(neq.right) == 'String'
		if (anyString) {
			return
'''(!«neq.left.compileExpression».equals(«neq.right.compileExpression»))'''
		}else{
			return neq.left.compileExpression + " == " +
neq.right.compileExpression
            }
      }

      static def CharSequence compileExpression(Expression exp) {
            switch (exp) {
                  And: exp.left.compileExpression + " && " +
exp.right.compileExpression
                  Or: exp.left.compileExpression + " || " +
exp.right.compileExpression
                  Equals: exp.properEquals
                  NotEquals: exp.properEquals
                  Greater: exp.left.compileExpression + " > " +
exp.right.compileExpression
                  Lesser: exp.left.compileExpression + " < " +
exp.right.compileExpression
                  GreaterOrEquals: exp.left.compileExpression + " >= " +
exp.right.compileExpression
                  LesserOrEquals: exp.left.compileExpression + " <= " +
exp.right.compileExpression
                  Concat: exp.left.compileExpression + " + " +
exp.right.compileExpression
                  Plus: exp.left.compileExpression + " + " +
exp.right.compileExpression
                  Minus: exp.left.compileExpression + " - " +
exp.right.compileExpression
                  Mult: exp.left.compileExpression + " * " +
exp.right.compileExpression
                  Div: exp.left.compileExpression + " / " +
exp.right.compileExpression
                  TypeValue: thisBinding(currentQuestion)
                  This: thisBinding(currentQuestion)
                  BooleanTrue: 'true'
                  BooleanFalse: 'false'
                  StringValue: "\"" + exp.value.toString() + "\""
                  NumberValue: exp.value.toString()
                  VariableUse: exp.ref.bindingName
                  Parenthesis: '(' + exp.exp.compileExpression + ')'
                  ExternalCall: 'external.' + exp.func.name + "(" +
String.join(",", exp.args.stream().map( x |
x.compileExpression).collect(Collectors.toList())) + ")"
                  Not: '!' + exp.exp.compileExpression
            }
      }

      static def toSnakeCase(String text) {
            var result = "";
            val starting = text.charAt(0)
            result += Character.toLowerCase(starting)
            for (var i = 1; i < text.length; i++) {
                  val character = text.charAt(i)
                  if (Character.isUpperCase(character)) {
                        result += "_" + Character.toLowerCase(character);
                  } else {
                        result += character;
                  }
            }
            return result;
      }
}
```

Scoping

```
/*
 * generated by Xtext 2.26.0
 */
package lomzt.mdsd.exam.scoping

import org.eclipse.emf.ecore.EObject
import org.eclipse.emf.ecore.EReference
import org.eclipse.xtext.scoping.Scopes
import static lomzt.mdsd.exam.Utilities.*
import lomzt.mdsd.exam.iF22.Story
import lomzt.mdsd.exam.iF22.Target
import lomzt.mdsd.exam.iF22.IF22Package
import lomzt.mdsd.exam.iF22.EndingTarget
import lomzt.mdsd.exam.iF22.Scenario
import org.eclipse.xtext.scoping.IScope

/**
 * This class contains custom scoping description.
 *
 * See
https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#scopi
```

```
ng
 * on how and when to use it.
 */
class IF22ScopeProvider extends AbstractIF22ScopeProvider {
      override getScope(EObject context, EReference reference) {

            if (context instanceof Target && reference ==
IF22Package.Literals.TARGET__TARGET) {
                  val scenarios = getParentOfType(context,
typeof(Story)).scenarios
                  return Scopes.scopeFor(scenarios, super.getScope(context,
reference))
            }

            if (context instanceof EndingTarget && reference ==
IF22Package.Literals.ENDING_TARGET__FROM) {
                  val scenario = getParentOfType(context,
typeof(Target)).target as Scenario
                  return Scopes.scopeFor(scenario.locations,
IScope.NULLSCOPE)
            }

            return super.getScope(context, reference)
      }
}
```

Validator

```
/*
 * generated by Xtext 2.26.0
 */
package lomzt.mdsd.exam.validation

import lomzt.mdsd.exam.iF22.Scenario
import org.eclipse.xtext.validation.Check
import lomzt.mdsd.exam.iF22.ExternalImport
import lomzt.mdsd.exam.iF22.Location
import lomzt.mdsd.exam.iF22.Expression
import lomzt.mdsd.exam.iF22.Question
import lomzt.mdsd.exam.iF22.This
import lomzt.mdsd.exam.iF22.Target
import lomzt.mdsd.exam.iF22.ExternalCall
import static lomzt.mdsd.exam.Utilities.*
import lomzt.mdsd.exam.iF22.Story
import lomzt.mdsd.exam.iF22.IF22Package
import lomzt.mdsd.exam.iF22.End
import lomzt.mdsd.exam.iF22.Type
import lomzt.mdsd.exam.iF22.EndingTarget
import lomzt.mdsd.exam.iF22.TypeValue

/**
 * This class contains custom validation rules.
 *
 * See
https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#valid
ation
 */
class IF22Validator extends AbstractIF22Validator {

      @Check
      def checkScenarioNameUnique(Scenario scenario) {
            val parent = getParentOfType(scenario, typeof(Story))
            for (os : parent.scenarios) {
                  if (os != scenario && os.name == scenario.name) {
                        error("Two scenarios may not have the same name.",
IF22Package.Literals.TARGETABLE__NAME)
                  }
            }
      }

      @Check
      def checkFunctionNameUnique(ExternalImport ei) {
            val parent = getParentOfType(ei, typeof(Story))
            for (oi : parent.imports) {
                  if (oi != ei && oi.name == ei.name) {
                        error("Two imports may not have the same name.",
IF22Package.Literals.EXTERNAL_IMPORT__NAME)
                  }
            }
      }

      @Check
      def checkLocationNameUnique(Location location) {
            val parent = getParentOfType(location, typeof(Scenario))
            for (otherLocation : parent.locations) {
                  if (otherLocation != location && otherLocation.name ==
location.name) {
                        error("Two locations may not have the same name.",
IF22Package.Literals.TARGETABLE__NAME)
                  }
            }
      }

      @Check
      def ensureAtLeastOneEndStatement(Scenario scenario) {
            val count = scenario.locations.stream().filter(x | x instanceof
End).count()
            if (count < 1) {
                  error ("Must have at least one end statement.",
IF22Package.Literals.SCENARIO__LOCATIONS)
            }
      }

      @Check
      def ensureTypeComparisonOnlyInInputValidation(TypeValue exp) {
            val question = getParentOfType(exp, typeof(Question))
            val children = getChildrenOfType(question.validate,
typeof(TypeValue))
            if (!children.contains(exp)) {
                  error("Type decleration is only valid within the context
of input validation", IF22Package.Literals.TYPE_VALUE__TYPE)
            }
      }

      @Check
      def ensureOnlyOneTypeInInputValidation(Question question) {
            val children = getChildrenOfType(question.validate,
TypeValue).size();
            if (children != 1) {
                  error("Must have only a single type declaration in input
validation.", IF22Package.Literals.QUESTION__VALIDATE)
            }
      }

      @Check
      def ensureThisOnlyInTargetExpressions(This obj) {
            val hasTargetParent = getParentOfType(obj, typeof(Target))
            if (hasTargetParent == false) {
                  error("'this' keyword can only be used within the context
of a target.", IF22Package.Literals.TARGET__CONDITION)
            }
      }

      @Check
      def ensureOnKeywordsOnlyWithExternalScenario(EndingTarget
endingTarget) {
            val target = getParentOfType(endingTarget, typeof(Target))
            if (!(target.target instanceof Scenario)) {
                  error("'on' keyword only applicable when moving to
another scenario.", IF22Package.Literals.ENDING_TARGET__FROM)
            }
      }

      @Check
      def ensureFunctionArgsMatchSignature(ExternalCall call) {
            val ip = call.func;
            if (call.func.params.size() != call.args.size()) {
                  error("Number of arguments does not match number of
parameters.", IF22Package.Literals.TARGET__ARGS)
            }
            for (var i = 0; i < ip.params.size(); i++) {
                  val expType = getExpressionOutputType(call.args.get(i))
                  val paramType = typeToJavaType(ip.params.get(i))
                  if (!expType.equals(paramType)) {
                        error("Arguments does not match function
signature.", IF22Package.Literals.EXTERNAL_CALL__ARGS)
                  }
            }
      }

      @Check
      def ensureScenarioArgsMatchSignature(Target target) {
            if (target.target instanceof Scenario) {
                  val scenario = target.target as Scenario;
                  if (scenario.params.size() != target.args.size()) {
                        error("Number of arguments does not match number of
parameters.", IF22Package.Literals.TARGET__ARGS)
                  }
                  for (var i = 0; i < scenario.params.size(); i++) {
                        val expType =
getExpressionOutputType(target.args.get(i))
```

```
                        val paramType =
typeToJavaType(scenario.params.get(i).type)
                        if (!expType.equals(paramType)) {
                              error("Arguments does not match scenario
signature.", IF22Package.Literals.TARGET__ARGS)
                        }
                  }
            }
      }

      @Check
      def ensureQuestionInputMatchVariableType(Question question) {
            if (question.binding != null) {
                  val questionInputType = inputType(question)
                  val bindingType = typeToJavaType(question.binding.type)
                  if (questionInputType !== bindingType) {
                        error("Cannot assign input of type " +
questionInputType + " to binding of type " + bindingType + ".",
IF22Package.Literals.QUESTION__BINDING)
                  }
            }
      }

      @Check
      def ensureValidationExpressionReturnBoolean(Question question) {
            if (!(question.validate instanceof TypeValue)) {
                  if (getExpressionOutputType(question.validate) !=
'boolean') {
                        error("Only boolean expressions are valid in input
validation.", IF22Package.Literals.QUESTION__VALIDATE)
                  }
            }
      }

      @Check
      def ensureTargetConditionReturnBoolean(Target target) {
            if (target.condition !== null) {
                  if (getExpressionOutputType(target.condition) !=
'boolean') {
                        error('Only boolean expressions are valid in the
transition condition.', IF22Package.Literals.TARGET__CONDITION)
                  }
            }
      }
}
```

Utilities

```
package lomzt.mdsd.exam

import java.util.ArrayList
import java.util.List
import lomzt.mdsd.exam.iF22.And
import lomzt.mdsd.exam.iF22.BooleanFalse
import lomzt.mdsd.exam.iF22.BooleanTrue
import lomzt.mdsd.exam.iF22.Concat
import lomzt.mdsd.exam.iF22.Div
import lomzt.mdsd.exam.iF22.Equals
import lomzt.mdsd.exam.iF22.Expression
import lomzt.mdsd.exam.iF22.ExternalCall
import lomzt.mdsd.exam.iF22.Greater
import lomzt.mdsd.exam.iF22.GreaterOrEquals
import lomzt.mdsd.exam.iF22.Lesser
import lomzt.mdsd.exam.iF22.LesserOrEquals
import lomzt.mdsd.exam.iF22.Minus
import lomzt.mdsd.exam.iF22.Mult
import lomzt.mdsd.exam.iF22.Not
import lomzt.mdsd.exam.iF22.NotEquals
import lomzt.mdsd.exam.iF22.NumberValue
import lomzt.mdsd.exam.iF22.Or
import lomzt.mdsd.exam.iF22.Parameter
import lomzt.mdsd.exam.iF22.Parenthesis
import lomzt.mdsd.exam.iF22.Plus
import lomzt.mdsd.exam.iF22.Question
import lomzt.mdsd.exam.iF22.StringValue
import lomzt.mdsd.exam.iF22.This
import lomzt.mdsd.exam.iF22.Type
import lomzt.mdsd.exam.iF22.VarBinding
import lomzt.mdsd.exam.iF22.VariableUse
import org.eclipse.emf.ecore.EObject
import lomzt.mdsd.exam.iF22.Text
import lomzt.mdsd.exam.iF22.TypeValue
```

```
class Utilities {

      // Adapted from semester project
      def static <T> getParentOfType (EObject obj, Class<T> clazz) {
            var cur = obj;
            while (cur.eContainer !== null) {
                  cur = cur.eContainer;
                  if (clazz.isInstance(cur)) {
                        return cur as T;
                  }
            }
            return null
      }

      def static <T> getChildrenOfType (EObject obj, Class<T> clazz) {
            val list = new ArrayList<T>()
            if (clazz.isInstance(obj)) {
                  list.add(obj as T);
            }
            getChildrenOfTypeRecursive(obj, clazz, list)
            return list
      }

      def static <T> void getChildrenOfTypeRecursive(EObject obj, Class<T>
clazz, List<T> list) {
            for (child : obj.eContents) {
                  if (clazz.isInstance(child)) {
                        list.add(child as T)
                  }
                  getChildrenOfTypeRecursive(child, clazz, list)
            }
      }

      def static CharSequence getExpressionOutputType(Expression exp) {
            switch (exp) {
                  And: 'boolean'
                  Or: 'boolean'
                  Equals: 'boolean'
                  NotEquals: 'boolean'
                  Greater: 'boolean'
                  Lesser: 'boolean'
                  GreaterOrEquals: 'boolean'
                  LesserOrEquals: 'boolean'
                  Concat: 'String'
                  Plus: 'int'
                  Minus: 'int'
                  Mult: 'int'
                  Div: 'int'
                  TypeValue: exp.type.typeToJavaType
                  This: getParentOfType(exp, typeof(Question)).inputType
                  BooleanTrue: 'boolean'
                  BooleanFalse: 'boolean'
                  StringValue: 'String'
                  NumberValue: 'int'
                  VariableUse: switch (exp.ref) {
                        VarBinding: (exp.ref as
VarBinding).type.typeToJavaType
                        Parameter: (exp.ref as
Parameter).type.typeToJavaType
                  }
                  Parenthesis: exp.exp.getExpressionOutputType
                  ExternalCall: exp.func.returnType.typeToJavaType
                  Not: 'boolean'
            }
      }

      static def typeToJavaType(Type type) {
            switch (type) {
                  lomzt.mdsd.exam.iF22.Number: 'int'
                  lomzt.mdsd.exam.iF22.Boolean: 'boolean'
                  Text: 'String'
            }
      }

      static def inputType (Question question) {
            val child = getChildrenOfType(question.validate,
typeof(Type)).get(0);
            switch (child) {
                  lomzt.mdsd.exam.iF22.Number: 'int'
                  lomzt.mdsd.exam.iF22.Boolean: 'boolean'
                  Text: 'String'
            }
      }
}
```