Translation of a Subset of RSL into Java

Ulrik Hjarnaa, s991422

12th November 2004

Preface

This thesis is the final requirement for obtaining the degree Master of Science in Engineering. The work was carried out at the Computer Science and Engineering section at the Institute of Informatics and Mathematical Modelling at the Technical University of Denmark. The duration of the project was from the 10th of May 2004 until the 12th of November 2004.

The project was supervised by Associate Professor, Ph.D. Anne E. Haxthausen and Associate Professor Hans Bruun.

I would like to thank my supervisors for constructive critisism and inspiration during the project. I would also like to thank the following people for comments on the report as well as proof reading: Martin W. Andersen and Poul-Henrik Worm.

Special thanks to my fiancée Ulla Berg Nielsen and my daughter Emma Berg Hjarnaa for support, understanding and patience throughout the project.

Lyngby, 11th November 2004

Ulrik Hjarnaa

Abstract

There exist a number of specification languages which are used for specifying the design and behaviour of software systems. Furthermore, a number of programming languages exist, which are used for implementation of software systems. This thesis focuses on a possible translation from the specification language RSL into the programming language Java.

This work identifies a translatable subset of the applicative part of RSL and gives suggestions for a translation of a subset of this into Java. The work focuses on the parts of RSL which have a direct translation in Java.

Based on these translations a prototype of a tool for carrying out the translation has been developed.

The tool has been developed using a combination of techniques. The front end has been developed using a tool for generating a lexer and a parser. The back end has been developed using the object–oriented visitor design pattern. The main part of the tool has been developed using a bootstrapping process. The main part of the tool was specified in RSL and translated using the tool itself.

Keywords: RAISE, RSL, Java, translation, bootstrapping.

Resumé

Der findes en række specifikationssprog som anvendes til at specificere designet af og egenskaber for software systemer. Derudover findes der en række programmeringssprog, som bliver anvendt til at implementere software systemer. Dette eksamensprojekt fokuserer på en mulig oversættelse fra specifikationssproget RSL til programmeringsproget Java.

Dette projekt identificerer en oversætbar delmængde af den applikative del af RSL og giver forslag til en oversættelse af en delmængde af denne til Java. Projektet fokuserer på de dele af RSL som har en direkte oversættelse i Java.

Baseret på disse oversættelser er der udviklet en prototype af et værktøj til at udføre af oversættelsen.

Værktøjet er blevet udviklet under anvendelse af en kombination af teknikker. Forenden er blevet udviklet under anvendelse af et værktøj til at generere en lexer og en parser. Bagenden er blevet udviklet under anvendelse af det objekt orienterede visitor design pattern. Hoveddelen af værktøjet er blevet udviklet under anvendelse af en bootstrapping proces. Hoveddelen af værktøjet blev specificeret i RSL og oversat med værktøjet selv.

Nøgleord: RAISE, RSL, Java, oversættelse, bootstrapping.

Contents

1 Introduction **1**
- 1.1 Thesis Objectives . . . 2
- 1.2 Requirements of the Developments . . . 2
- 1.3 Related Work . . . 3
- 1.4 Prerequisites . . . 3
- 1.5 Report Overview . . . 4

2 Informal Description of Translation **5**
- 2.1 Translation from RSL into Java . . . 5
  - 2.1.1 Identifiers in RSL and Identifiers in Java . . . 5
- 2.2 Top Level Structure of RSL . . . 6
  - 2.2.1 Class Expressions . . . 6
  - 2.2.2 Objects and Schemes . . . 7
  - 2.2.3 Operators on Class Expression . . . 9
  - 2.2.4 Parameterized Schemes . . . 13
  - 2.2.5 Object Arrays . . . 17
  - 2.2.6 Overview of translation of top–level structures . . . 18
- 2.3 Types in RSL . . . 22
  - 2.3.1 Built–in Types . . . 22
  - 2.3.2 Complex Types . . . 29
  - 2.3.3 Type Definitions . . . 42
  - 2.3.4 Subtypes . . . 54
- 2.4 Values in RSL . . . 55
  - 2.4.1 Ordering of Variables and Values . . . 55
  - 2.4.2 Value Definitions . . . 57
  - 2.4.3 Value Expressions . . . 61
- 2.5 Variable Definitions . . . 73
- 2.6 Axiom Definitions . . . 73
- 2.7 Channel Definitions . . . 73
- 2.8 Test Cases Definitions . . . 73

3 Strategy **75**
3.1 Design Options . . . 75
3.1.1 Bootstrapping . . . 75
3.1.2 Design Patterns . . . 82
3.2 Architechtural Design of the Translator . . . 86
3.2.1 Data Structures . . . 88
3.2.2 Front End of the translator . . . 90
3.2.3 Translation Module . . . 92
3.2.4 Back End of the translator . . . 96
3.2.5 Control Module . . . 96
3.3 Development Process . . . 97
3.4 Subsets of RSL . . . 101
3.4.1 Description of RSL_0 . . . 101
3.4.2 Description of RSL_1 . . . 104

4 Implementation of the Translator **105**
4.1 The Library Classes . . . 105
4.2 First Version of the Translator . . . 109
4.2.1 AST Representation of RSL . . . 109
4.2.2 AST Representation of Java . . . 111
4.2.3 Front End . . . 115
4.2.4 Translation Module . . . 116
4.2.5 Back End . . . 122
4.2.6 Control Module . . . 124
4.3 Second Version of the Translator . . . 125
4.3.1 AST Representation of RSL . . . 125
4.3.2 AST Representation of Java . . . 126
4.3.3 Front End . . . 126
4.3.4 Translation Module . . . 126
4.3.5 Back End . . . 131
4.3.6 Control Module . . . 131

5 Overview of the Formal Specification **133**
5.1 Abstract Syntax Tree for RSL . . . 134
5.2 Abstract Syntax Tree for Java . . . 136
5.3 Wrapper Modules . . . 136
5.4 Translation Module . . . 138

6 Test **141**
6.1 Purpose . . . 141
6.2 Test Schemes . . . 141

6.3 Results 143

7 Extensions and Further Work **145**
7.1 Creating Tools Transforming RSL 145
7.2 Expanding the Translation 147
7.3 Ideas for Translation 148
7.3.1 Variable Declarations 148
7.3.2 Channel Declarations 149
7.3.3 Parallel Composition and External Choice 149
7.3.4 Pre–conditions 150
7.3.5 Let Expressions 151
7.3.6 Case Expressions 151
7.3.7 For Expressions 153
7.3.8 While Expressions 153
7.3.9 Until Expressions 154
7.4 Translating Larger Subsets of RSL 154

8 Conclusion **157**
8.1 Results 157
8.2 Concluding Remarks 158

A Using the Translator and Extending the Translator **161**
A.1 Using the Translator 161
A.1.1 Tools Needed 161
A.1.2 Setting up the Translator 161
A.1.3 Controlling the Translation Process 162
A.2 Extending the Translator 163
A.2.1 Tools Needed 163
A.2.2 Extend the Front End 163
A.2.3 Extend the Translation Module 164
A.2.4 Extend the Back End 165

B Content of CD **167**

C Formal Specification **169**
C.1 First Specification, Intermediate Version 169
C.1.1 Translator_Module 169
C.1.2 RSLAst_Module 219
C.1.3 RSLAst_WrapperModule 222
C.1.4 RSLAst_WrapperModule_2 222
C.1.5 JavaAst_Module 224

C.2 Second Version . . . 228
C.2.1 Translator_Module2 . . . 228
C.2.2 RSLAst_Module2 . . . 278
C.2.3 RSLAst_WrapperModule2 . . . 282
C.2.4 RSLAst_WrapperModule_22 . . . 282
C.2.5 JavaAst_Module2 . . . 284

D ANTLR Grammars 289
D.1 Grammar file for the first version . . . 289
D.2 Grammar file for the second version . . . 309

E Source Code, First Version 331
E.1 translator . . . 331
E.1.1 Translator.java . . . 331
E.2 translator.rsllib . . . 388
E.2.1 RSLList.java . . . 388
E.2.2 RSLListDefault.java . . . 389
E.2.3 RSLSet.java . . . 392
E.2.4 RSLSetDefault.java . . . 392
E.2.5 RSLMap.java . . . 394
E.2.6 RSLMapDefault.java . . . 395
E.3 translator.rslast . . . 397
E.3.1 ApplicationExpr.java . . . 397
E.3.2 BasicClassExpr.java . . . 398
E.3.3 Binding.java . . . 399
E.3.4 CaseBranch.java . . . 399
E.3.5 CaseExpr.java . . . 400
E.3.6 ClassExpr.java . . . 401
E.3.7 ComponentKind.java . . . 402
E.3.8 Constructor.java . . . 403
E.3.9 Decl.java . . . 404
E.3.10 Destructor.java . . . 404
E.3.11 DisambiguationExpr.java . . . 405
E.3.12 ElseBranch.java . . . 406
E.3.13 ElsifBranch.java . . . 406
E.3.14 EnumeratedListExpr.java . . . 407
E.3.15 ExplicitFunctionDef.java . . . 408
E.3.16 ExtendingClassExpr.java . . . 410
E.3.17 FiniteListTypeExpr.java . . . 411
E.3.18 FormalFunctionApplication.java . . . 411
E.3.19 FormalFunctionParameter.java . . . 412

E.3.20 FunctionArrow.java . . . 413
E.3.21 FunctionResultDescription.java . . . 413
E.3.22 FunctionTypeExpr.java . . . 415
E.3.23 Id.java . . . 416
E.3.24 IdApplication.java . . . 417
E.3.25 IdOrOp.java . . . 418
E.3.26 IfExpr.java . . . 418
E.3.27 InfixExpr.java . . . 420
E.3.28 LibModule.java . . . 420
E.3.29 ListExpr.java . . . 421
E.3.30 ListTypeExpr.java . . . 422
E.3.31 NamePattern.java . . . 422
E.3.32 NoAccessDescription.java . . . 423
E.3.33 NoPrecondition.java . . . 423
E.3.34 OptionalAccessDescription.java . . . 424
E.3.35 OptionalPrecondition.java . . . 424
E.3.36 Pattern.java . . . 424
E.3.37 Precondition.java . . . 424
E.3.38 PrefixExpr.java . . . 425
E.3.39 ProductTypeExpr.java . . . 425
E.3.40 RecordPattern.java . . . 426
E.3.41 RecordVariant.java . . . 427
E.3.42 RSLAst.java . . . 428
E.3.43 RSLInfixOp.java . . . 429
E.3.44 RSLOp.java . . . 430
E.3.45 RSLPrefixOp.java . . . 430
E.3.46 SchemeDef.java . . . 431
E.3.47 SchemeInstantiation.java . . . 432
E.3.48 ShortRecordDef.java . . . 432
E.3.49 SingleTyping.java . . . 434
E.3.50 SortDef.java . . . 434
E.3.51 StructuredExpr.java . . . 435
E.3.52 TestDecl.java . . . 435
E.3.53 TestDef.java . . . 436
E.3.54 TypeDecl.java . . . 437
E.3.55 TypeDef.java . . . 438
E.3.56 TypeExpr.java . . . 438
E.3.57 TypeLiteral.java . . . 438
E.3.58 TypeName.java . . . 440
E.3.59 ValueDecl.java . . . 440
E.3.60 ValueDef.java . . . 441

E.3.61 ValueExpr.java . . . 441
E.3.62 ValueInfixExpr.java . . . 442
E.3.63 ValueLiteral.java . . . 443
E.3.64 ValueLiteralBool.java . . . 443
E.3.65 ValueLiteralChar.java . . . 444
E.3.66 ValueLiteralInteger.java . . . 444
E.3.67 ValueLiteralPattern.java . . . 445
E.3.68 ValueLiteralReal.java . . . 445
E.3.69 ValueLiteralText.java . . . 446
E.3.70 ValueOrVariableName.java . . . 446
E.3.71 ValuePrefixExpr.java . . . 447
E.3.72 Variant.java . . . 448
E.3.73 VariantDef.java . . . 448
E.3.74 WildcardPattern.java . . . 449
E.4 translator.javaast . . . 450
E.4.1 ArrayCreation.java . . . 450
E.4.2 ArrayType.java . . . 451
E.4.3 AssignmentExpression.java . . . 452
E.4.4 AssignmentOperator.java . . . 453
E.4.5 Block.java . . . 454
E.4.6 BooleanLiteral.java . . . 455
E.4.7 CastExpression.java . . . 456
E.4.8 CharLiteral.java . . . 457
E.4.9 ClassDeclaration.java . . . 457
E.4.10 ClassInstanceCreation.java . . . 461
E.4.11 CompilationUnit.java . . . 462
E.4.12 ConstructorDeclaration.java . . . 464
E.4.13 DoubleLiteral.java . . . 466
E.4.14 Expression.java . . . 467
E.4.15 ExpressionStatement.java . . . 467
E.4.16 FieldAccessExpression.java . . . 468
E.4.17 FieldDeclaration.java . . . 469
E.4.18 IfStatement.java . . . 470
E.4.19 ImportDeclaration.java . . . 472
E.4.20 InfixExpression.java . . . 472
E.4.21 InfixOperator.java . . . 473
E.4.22 InstanceOfExpression.java . . . 474
E.4.23 IntegerLiteral.java . . . 475
E.4.24 JavaAst.java . . . 476
E.4.25 JavaName.java . . . 477
E.4.26 JavaType.java . . . 477

E.4.27 MethodDeclaration.java . . . 478
E.4.28 MethodInvocation.java . . . 480
E.4.29 Modifier.java . . . 482
E.4.30 NoPackageDeclaration.java . . . 482
E.4.31 NullLiteral.java . . . 483
E.4.32 OptionalPackageDeclaration.java . . . 483
E.4.33 PackageDeclaration.java . . . 484
E.4.34 ParentherizedExpression.java . . . 484
E.4.35 PrefixExpression.java . . . 485
E.4.36 PrefixOperator.java . . . 486
E.4.37 PrimitiveType.java . . . 486
E.4.38 QualifiedName.java . . . 487
E.4.39 ReferenceType.java . . . 488
E.4.40 ReturnStatement.java . . . 489
E.4.41 SimpleName.java . . . 490
E.4.42 SingleVariableDeclaration.java . . . 491
E.4.43 Statement.java . . . 492
E.4.44 StringLiteral.java . . . 493
E.4.45 ThisExpression.java . . . 493
E.4.46 TypeDeclaration.java . . . 494
E.4.47 VariableDeclaration.java . . . 494
E.4.48 VariableDeclarationExpression.java . . . 495
E.4.49 VariableDeclarationFragment.java . . . 496
E.4.50 VariableDeclarationStatement.java . . . 497
E.5 translator.lib . . . 499
E.5.1 AST.java . . . 499
E.5.2 Element.java . . . 499
E.5.3 JavaElement.java . . . 499
E.5.4 JavaVisitor.java . . . 500
E.5.5 ParentVisitor.java . . . 506
E.5.6 RSLElement.java . . . 513
E.5.7 StringJavaVisitor.java . . . 514
E.5.8 StringVisitor.java . . . 524
E.5.9 TypeDecorateVisitor.java . . . 534
E.5.10 TypeEvaluator.java . . . 551
E.5.11 Visitor.java . . . 552

F Source Code, Second Version 559
F.1 translator . . . 559
F.1.1 Runner2.java . . . 559
F.2 translator.lib . . . 565

F.2.1 TM_JavaAstVisitor.java . . . 565
F.2.2 TM_ParentRSLAstVisitor.java . . . 575
F.2.3 TM_RSLAstVisitor.java . . . 582
F.2.4 TM_RSLElement.java . . . 590
F.2.5 TM_StringJavaAstVisitor.java . . . 591
F.2.6 TM_StringRSLAstVisitor.java . . . 605
F.2.7 TM_TypeDecorateRSLAstVisitor.java . . . 615

G Test Code **641**
G.1 Test of library classes . . . 641
G.1.1 RSLListDefaultTest.java . . . 641
G.1.2 RSLSetDefaultTest.java . . . 643
G.1.3 RSLMapDefaultTest.java . . . 644

H Test Results **647**
H.1 Library classes . . . 647
H.1.1 RSLListDefaultTest . . . 647
H.1.2 RSLLSetDefaultTest . . . 648
H.1.3 RSLMapDefaultTest . . . 648

I Examples **651**
I.1 LISTSUM . . . 651
I.1.1 LISTSUM.rsl . . . 651
I.1.2 LISTSUM.java . . . 652
I.1.3 Result of Execution . . . 653
I.2 LIST . . . 654
I.2.1 LIST.rsl . . . 654
I.2.2 List.java . . . 654
I.2.3 MyList.java . . . 655
I.2.4 Empty.java . . . 655
I.2.5 Add.java . . . 656
I.2.6 Result of Execution . . . 656
I.3 APPLICATION . . . 657
I.3.1 APPLICATION.rsl . . . 657
I.3.2 APPLICATION.java . . . 657
I.3.3 MyShortRecordDefinition.java . . . 657
I.3.4 Result of Execution . . . 658

List of Tables

2.1 Operators and connectives for type **Bool** 24
2.2 Overview of Java types for representing integer values. 25
2.3 Operators for the type **Int** 26
2.4 Overview of Java types for representing real values. 26
2.5 Overview of translations of **Text** operations in Java 28
2.6 Summary of translation of built–in types in RSL. 28
2.7 Overview of translation of list operators in RSL to Java. . . . 35
2.8 Overview of translation of set operators in RSL to Java. . . . 38
2.9 Overview of translation of RSL map operators into Java. . . . 42

List of Figures

3.1 A program P implemented in C. 76
3.2 A machine running machine code M. 77
3.3 An interpreter interpreting P implemented in I. 77
3.4 A translator translating from S into T implemented in I. . . . 77
3.5 A Java program P being translated to Java Byte Code (JBC) using a Java to JBC translator implemented in C. 78
3.6 An RSL to Java translator implemented in Java being translated into a RSL to Java translator implemented in JBC, using a Java to JBC translator implemented in JBC interpreted by a JBC to M interpreter on a machine M. 78
3.7 A two–stage compiler compiling from Java to M with JBC as intermediate language. 79
3.8 A compiler for Java_0 into JBC implemented in C. 79
3.9 Compilation of the developed compiler. 80
3.10 Second version of the compiler written in Java_s. 80
3.11 Compilation of the second version of the compiler. 80
3.12 Third version of the compiler extended to the whole source language. 81
3.13 Compilation of the third version of the compiler. 81
3.14 Left: Translator implemented in the source language. Right: Translator implemented in the target language. 82
3.15 The modified translator producing high quality target language. 82
3.16 Translation of the new version of the translator using the first version. 82
3.17 Translation of the second version of the translation using the second version. 83
3.18 Questionnaire–structure without the visitor design pattern. . . 85
3.19 Questionnaire–structure with the visitor design pattern. 85
3.20 Flow of translation. 89
3.21 Lexical analysis of if–then–else statement in RSL. 90
3.22 Parsing of if–then–else statement in RSL. 90

3.23 Using ANTLR. 97
3.24 Front end: Tool translating from RSL_0 to $RSLAST_0$. 98
3.25 Translation Module: Tool translating from $RSLAST_0$ to JavaAst. 98
3.26 Back end: Tool for generating Java from the JavaAst. 98
3.27 Combining the parts to a translator from RSL_0 to Java. 99
3.28 Result of combining the parts. 99
3.29 Specification of a new translation module. 100
3.30 Translating the specification using the first version of the translator. 100
3.31 Combining the new translation module with the front end and back end. 100
3.32 Result of combining the new parts. 100

7.1 Tool translating between two kinds of RSL AST's implemented in RSL 146
7.2 Translating the tool specified. 146
7.3 Combining the new tool developed with the front end and the RSL visitor as back end. 146

List of examples

2.1 An under–specified basic class expression in RSL. 7
2.2 Example of declarations of objects in RSL. 8
2.3 Example of use of an object within another object. 8
2.4 Using the extend feature in Java for extension in RSL. 11
2.5 Translation of a scheme not hiding the variable. 12
2.6 Translation of a scheme hiding the variable. 12
2.7 Renaming class expression. 13
2.8 Parameterized list in RSL. 14
2.9 Translation of a parameterized list in Java. 14
2.10 Instantiation of parameterized scheme. 15
2.11 Expanded instantiated parameterized scheme equivalent. . . . 16
2.12 Translation of parameterized scheme presented in Example 2.10. 17
2.13 Translation of an object array declaration. 18
2.14 Translation of a product. 30
2.15 Defining a function using an interface. 32
2.16 Defining an uncurried function using an interface. 33
2.17 List creation – examples. 35
2.18 List operators – examples. 36
2.19 Set creation - examples. 38
2.20 Use of infix operators on sets. 39
2.21 Creation of maps. 41
2.22 Use of map operators. 43
2.23 Structural equivalence in RSL 44
2.24 Recursive variant type definition 49
2.25 Translation of variant type definition using an enumeration. . 50
2.26 Variant type definition having constructors, destructors and reconstructors 50
2.27 Not recursive variant definition. 51
2.28 Translation of a short record definition. 53
2.29 Abbreviation definition 54
2.30 Specification applying a maximal type to a function of a subtype. 55

2.31 Translation of a subtype using assertions. . . . 56
2.32 Ordering problem for values and variables. . . . 56
2.33 Correct ordering for the field in Java. . . . 57
2.34 Listsum translation. . . . 59
2.35 Listsum translation closest to specification 59
2.36 Translation of explicit value definitions. . . . 61
2.37 Not allowed translation of nested **if–then–else** expression. . . 62
2.38 Allowed translation of nested **if–then–else** expression. . . . 62
2.39 Rewrite elsif construct. . . . 64
2.40 Translation of a case expression using name patterns. . . . 66
2.41 Translation of a case expression using record and name patterns. 67
2.42 Translation of a case expression using value literal patterns and a wildcard pattern. . . . 68
2.43 Translation of record constructors, make function, destructors and reconstructors. . . . 70
2.44 Translation value infix expressions. . . . 72
2.45 Translation of a test case. . . . 74
3.1 Visitor pattern – traversal in object structure. . . . 86
3.2 Visitor pattern – traversal in visitor. . . . 87
3.3 Different kinds of application expressions in RSL. . . . 93
3.4 LISTMULT example. . . . 95
4.1 Generic types, boxing and unboxing in Java 1.5. . . . 106
4.2 `hd` and `tl` methods in class: *RSLListDefault* 107
4.3 compose method in class: *RSLMapDefault* 108
4.4 Syntax rule *type_def* and corresponding Java implementation. 110
4.5 Implementation of non–terminal containing other non–terminals. 111
4.6 Implementation of non–terminal containing only terminal symbols. . . . 112
4.7 Non–terminal which consists of several alternatives, which are non–terminals. . . . 113
4.8 Non–terminal containing several non–terminals. . . . 113
4.9 Non–terminal which consists of several alternatives, which are all terminals. . . . 114
4.10 Rules for different precedence levels of operators. . . . 117
4.11 Front end of the first version. . . . 118
4.12 Top level function of translation module. . . . 118
4.13 Use of recursion in the translation module. . . . 119
4.14 *makeVisitorMethod* method. . . . 120
4.15 Illustration of difference between implementation and idea presented in section 2.4.3. . . . 123
4.16 Visit method for if–else statement. . . . 124

4.17 Difference in implementation and specification of prefix operators in RSL. 127
4.18 Differences in creation of an if statement between the first and second version of the translator. 130
5.1 Implementation of TM_ValueExpr and TM_VariableOrValueName135
5.2 Rule: *prefix_op* RSL: TM_PrefixOperator. 136
5.3 Simplifying a hierachy. 137
7.1 Translation of a function using a pre–condition. 150
7.2 Idea of translation of list pattern in a case expression. 152
7.3 Idea of translation of an equality pattern in a case expression. 152
7.4 Translation of a for–loop. 153

Notation Guide

Overall notation will attempt to follow well established guidelines where possible. The following conventions have been used. Single keywords or very short examples of code:

RSL keywords are in ordinary type face written: **value**.

RSL specifications are in ordinary type face written: `function`.

RSL Scheme and Object names are in ordinary type face written: MY_MODULE.

Java keywords and names are in ordinary type face written: *Object*.

Java code is in ordinary type face written: equals.

ANTLR grammar is in ordinary type face written: `scheme_def`.

Definitions of translations are written as an RSL specification followed by a listing of Java source code both of which may include mathematical symbols for indexing identifiers. Examples of translations, longer specifications in RSL and listings of Java source code are written as numbered examples.

Chapter 1

Introduction

In the process of developing software, there are several methods or approaches. These methods range from rather ad hoc ones to methods which are very systematic and well organized. Each of the approaches has a number of strengths and weaknesses. In the category of systematic methods there are several kinds, some of which are visually based, and some based on mathematics. Most of the mathematically based methods for software development have an accompanying language or notation for writing specifications of software.

Steps in a mathematically based method include writing a specification in the accompanying language or notation and implementing the specification in a programming language. It is the similarities between the final specification and the implementation which are of interest in this project. When comparing implementations of different specifications there are often a number of constructs in the specification language, which are implemented in the same way in the programming language.

This leads to the idea that a specification language or at least a subset of a specification language can be translated systematically to a programming language. If it is possible to define a systematic translation, then it may be possible to develop a tool for carrying out the translation.

This project focuses on translation from the formal specification language (RSL) RAISE Specification Language into the programming language Java.

RAISE is a formal method for software development. It is an abbreviation for Rigorous Approach to Industrial Software Engineering. RAISE consists of three parts:

- A method for software development: RAISE [12].
- A specification language: RSL[11].

- A set of tools for the specifications: rsltc [10], eden [15].

RSL is a wide spectrum specification language based on mathematics. It is a high level language, and it offers many means for specifying the behaviour of software. RSL offers posibilities of writing applicative specifications, algebraic specification and imperative specifications.

Java is a general purpose imperative programming language. It is a high level language meaning that complicated operations may be expressed using simple statements and predefined objects and methods. Java is aimed at being portable between different hardware platforms: "Write once, run everywhere". The idea of being highly portable makes it well suited for an automatic translation, since the translation does not have to deal with different platforms. The source code generated should be able to run on all machines, for which Java is available.

Some specification styles in RSL are not well suited for implementation in a programming language. There are two issues which seems problematic in a translation to a programming language. In RSL, it is possible to underspecify the values in a specification, which makes it impossible to determine the exact value. Furthermore, the syntax of RSL allows for implicit definitions, which makes it very difficult to determine a value.

In general these implicit ways of specification cannot be implemented in an imperative programming language, because in an imperative programming language everything must be specified explicitly. This is the reason why only a subset of RSL can be translated into Java.

1.1 Thesis Objectives

The idea presented above that a subset of the specification language RSL can be translated systematically to the programming language Java, has lead to the following objectives:

1. Definition of a *translation* from a subset of RSL into Java.
2. Development of a tool for translating from RSL into Java, called a *translator*.

1.2 Requirements of the Developments

The two objectives listed above have lead to a number of requirements for each of them. These requirements are described and discussed in the following.

Requirements of Translation

1. The translation from RSL into Java must be semantically equivalent to the specification in RSL.

2. In the Java translated from the specification it should be possible to recognize the specification.

The semantics of the languages are not formally specified in a way that makes a formal proof of the semantical equivalence possible. The first correctness requirement of the translation can therefore only be argued informally. The second requirement is to ensure that it is possible to integrate a part of a system which is specified and translated using a tool, and other parts of a system which are implemented without the use of a tool.

Requirements of Translator

1. The translator must translate a subset of RSL into Java according to the translation defined.

2. The translator must provide extensibility.

The requirement of being open to extensions is set to ensure that if a translation for a larger subset of RSL is defined, then it should be easy to extend the translator to translate this new subset.

1.3 Related Work

There are a number of tools developed for RSL [15, 10], including a type–checker, a pretty–printer and translators from RSL into a number of different programming languages. Most of these tools have been developed using the Gentle Compiler Construction System [17] or Cornell Syntesizer Generator [16]. The type–checker has been used in this project to ensure that specifications have the correct syntax before they are translated into Java.

1.4 Prerequisites

This report has a number of prerequisites for reading it.

1. A knowledge of RSL.

2. A knowledge of Java.

3. A basic idea of how to develop a language processor.

The meaning and use of the different constructs in the two languages are not explained. It is assumed that the reader is familiar with both the meaning and the use of the constructs. The process of developing a language processor is explained but some knowledge of the key concepts is expected.

1.5 Report Overview

The remainder of the report is structured as follows. Chapter 2 develops a general idea of how to define a translation from the specification language RSL into the programming language Java. Based on the ideas in Chapter 2 the development strategies for a translator are discussed in Chapter 3, which also outlines some of the technical issues involved in developing a translation between the languages. Chapter 4 and 5 describes the actual implementation of the translator. Chapter 6 describes the test schemes used for testing the tool developed in this project. Chapter 7 gives suggestions for extensions of the work done in this project. Finally, Chapter 8 summarizes the achievements of this work and evaluates these.

Chapter 2

Informal Description of Translation

This chapter presents an informal description of the translation from RSL into Java. The discussion will go through the constructs in RSL. For each of the constructs a discussion of the possible translations considered are given, as well as a description of the translation chosen.

The chapter starts with some considerations concerning a number of issues which are not related to a specific construct in RSL, followed by a section dealing with the top–level structures of RSL. Finally, there is a section covering each kind of declarations in RSL.

2.1 Translation from RSL into Java

Even though it is possible to find many correspondences between RSL and Java and to define translations for many of the constructs in RSL there are some issues, that do not belong to any individual construct in RSL, but which are of a more general nature. These general issues are covered in the following.

2.1.1 Identifiers in RSL and Identifiers in Java

In RSL, identifiers are used in numerous places e.g. for naming types and values. In Java, there are also identifiers which are used for naming fields, methods and classes. In the translation from RSL into Java it would be an advantage for the recognizability between the specification and the translation, if the identifiers in RSL could be reused in the translation. Two things should be kept in mind:

1. Differences in the character sets allowed for identifiers.

2. Differences in where the same identifier may be used again.

In RSL, an identifier must start with an ASCII letter or a greek letter, and it may contain ASCII letters, greek letters, digits, _ and ‘’. An RSL identifier may not start with _. Identifiers starting with _ are permitted in Java and this can be used to avoid name clashes when defining auxiliary identifiers in Java. Identifiers in RSL cannot be reused in Java without consideration of what to do with the characters which are allowed in RSL identifiers, but not in Java. To make the translation easy and to make it easier to see a correspondence between the identifiers in the specification and in the translation, only identifiers using the English character set, digits, and underscore are considered in this project.

The scope of identifiers are different in Java compared to RSL. The structure of the two languages are completely different. The translation of types from RSL into Java is performed using classes. Two classes may not be named identically within a package in Java, therefore there cannot be two types using the same name in the specification even though they belong to different scopes in RSL. In this work, reuse of identifiers is not considered.

2.2 Top Level Structure of RSL

The top level structure of an RSL specification is a module. A module may contain an entire specification as well as other modules, and it is therefore possible to decompose a large specification into smaller comprehensible and reusable units. There exist two kinds of modules, namely *objects* and *schemes*. Both of these structures rely on the concept: A class expression. This concept is presented in the following and afterwards the two kinds of modules are discussed.

2.2.1 Class Expressions

The most fundamental form of a class expression is a *basic class expression*. There are a number of operators for manipulating a class expression, but the only way to create a class expression is by a basic class expression. A basic class expression contains a list of declarations. In Example 2.1 a basic class expression containing a value declaration and an axiom declaration is shown. The meaning of a class expression is essentially a set of models. The reason that it is a set of models is that it is possible to under–specify the declarations. The class expression presented in Example 2.1 contains two

models, which satisfy the declarations, namely one where i is bound to 1 and one where i is bound to 2.

Example 2.1 An under–specified basic class expression in RSL.

```
class
  value i : Int
  axiom i = 1 ∨ i = 2
end
```

A class expression may be named and instantiated by using objects and schemes.

2.2.2 Objects and Schemes

An object is essentially a named model chosen from a class of models represented by some class expression [11]. A scheme, on the other hand, is a naming of a class expression. One could say that an object is an instance of a class expression. The concept of a scheme, i.e. a naming of a class, is necessary in order to be able to manipulate class expressions without having to write the entire class expression multiple times.

The idea of grouping a number of values, variables and functions can also be found in Java, namely the *class* construct in Java. A class in Java is able to hold values, variables and methods and it is therefore chosen as translation of the basic class expression. In Java, a class must be given a name. In RSL, a class expression is not named, but a scheme is a named class expression, and therefore, the translation of a scheme in RSL is a class in Java. The use of operators on a class expression does not change the basic idea that a scheme is translated as a class. A scheme may be parameterized allowing for a specification of a more general solution and then concretized in instantiations. The use of parameterization of schemes complicates the process, but this does not change the idea, that a scheme is a named class expression, and therefore should be translated as some sort of class in Java. A discussion of parameterized schemes can be found in Section 2.2.4

An object can be considered as an instance of a class expression in RSL. An object is declared by the entire class expression after the name of the object. Two forms of declaration of an object are shown in Example 2.2.

An object in RSL may be referenced by other modules. In Example 2.3, a third object referencing the objects (LIST1, LIST2) from Example 2.2 is shown.

Example 2.2 Example of declarations of objects in RSL.

```
scheme LIST_SCHEME =
  class
    variable list : Int*
    value
      empty : Unit → write list Unit
      empty() ≡ list := ⟨⟩,
      is_empty : Unit → read list Bool
      is_empty() ≡ list = ⟨⟩,
      add : Int → write list Unit,
      add(i) ≡ list := ⟨i⟩ ⌢ list,
      head : Unit ~→ read list Int
      head() ≡ hd list
      pre ~is_empty(),
      tail : Unit ~→ write list Unit
      tail() ≡ list := tl list
      pre ~is_empty()
  end
object
  LIST1 : LIST_SCHEME,
  LIST2 : LIST_SCHEME
/*They could also be declared as*/
object
  LIST1 : class ... content of class expression in LIST_SCHEME ... end
  LIST2 : class ... content of class expression in LIST_SCHEME ... end
```

Example 2.3 Example of use of an object within another object.

```
object LIST_APPLY :
  class
    value
      apply : (Int → Int) → read LIST1.list
      apply(f) ≡
        if ~LIST1.is_empty() then
          let first = LIST2.head() in
            LIST2.tail() ; apply(f) ; LIST2.add(f(first))
          end
        end
  end
```

The two examples of declarations of objects presented give two possible translations in Java:

1. Translate each object and scheme as a class of its own with static variables and static methods.

2. Translate the scheme LIST_SCHEME as a class and let the two objects LIST1 and LIST2 be translated as instances of the class which results of the translation of LIST_SCHEME if the first form of declaration of objects is used. LIST_APPLY should then be translated as an instance of another class and the same goes for LIST1 and LIST2 if the second form of declaration of objects is used.

As described in Section 2.1.1 an identifier in RSL cannot start with an underscore character this can be used to create a class for the object LIST_APPLY and then let LIST_APPLY be an instance of this class.

Section 2.2.6 discusses the advantages and disadvanteges of the two possible solutions.

2.2.3 Operators on Class Expression

In RSL, there are three ways of manipulating a class expression:

1. Extension.

2. Hiding.

3. Renaming.

Common for these three operators are that they all result in a new class expression, which may be manipulated further.

Extension

The idea of extension in RSL is to build a class expression in successive steps, adding new properties at each step. The form of an extending class expression is:

extend base_class **with** extending_class

The extending class expression may use all the declarations defined in the base class expression. In fact, the extending class expression is equivalent to a basic class expression containing all the declarations of both `base_class` and `extending_class`. Both `base_class` and `extending_class` may be any kind

of class expression including an identifier of another scheme, this is called a scheme instantiation. The most common use of an extending class expression is to let the base class be an identifier of a scheme defined elsewhere, and let `extending_class` be a basic class expression containing declarations of the new properties.

The way the extension is defined in RSL gives two possibilities for translation into Java:

1. Use the fact that an extending class expression in RSL is equivalent to a basic class expression, and translate the basic class expression
2. Use the possibility in Java to extend one class with another.

Both of these suggestions have advantages and disadvantages. The disadvantage of the first suggestion is, that the Java source code generated is not divided into several classes. It is harder to read and not as close to the idea in the specification, which is to have the system divided into several smaller units. The advantage of the first suggestion is that it is easy to deal with since all that is needed is a translation of all the declarations of the two class expressions. The advantage of the second suggestion is that the idea of the specification to split the details of the specification into smaller units is preserved. This property is kept by creating one class in Java per class expression in RSL, and let the classes created extend each other in the right order, as shown in Example 2.4. The disadvantage of the second suggestion is that it is more complicated to deal with than one large basic class expression.

The second suggestion is chosen because it preserves the idea of dividing a specification into several schemes, and keeping this as several pieces in Java.

Hiding

The purpose of the hide operator in RSL is to hide identifiers used inside a class expression from modules using the class expression either by extension or by instantiation. The form of a hiding class expression is:

hide id **in** class_expression

In Java, identifiers must be declared public to be accessible from classes in other packages. In RSL, all declarations are accessible by other modules unless they are explicitly hidden, therefore, as default all fields and methods are declared *public* in the Java source code generated. When the hide expression is used the access modifier of a field or method is changed to *protected* to disallow access, as shown in Examples 2.5 and 2.6. As stated later type definitions should be translated as classes in Java. These classes are by default

Example 2.4 Using the extend feature in Java for extension in RSL.

```
scheme LIST_STATE =
  class
    variable list : Int*
  end


scheme LIST_OPERATIONS =
  extend LIST_STATE with
  class
    value
      is_empty : Unit → read list Bool
      is_empty() ≡ list = ⟨⟩
  end
```

```
public class LIST_STATE {
    public static RSLListDefault<Integer> list;
    //RSLListDefault is explained in Section 2.3.2.
}

public class LIST_OPERATIONS extends LIST_STATE {
    public static boolean is_empty() {
        return list.equals(new RSLListDefault<Integer>());
    }
}
```

declared *public* and put into a file of their own. If a type is to be hidden, the corresponding class should be declared *private*. A *private* class must be declared within the class that uses it. This presents a problem because in a specification more than one type definition may use a hidden type definition as a type of a component. When these type definitions are translated as different *public* classes in different files, then there is no place where the *private* class corresponding to the hidden type can be declared so that both *public* classes can access it. Hiding of types should therefore be disallowed in the translation.

Example 2.5 Translation of a scheme not hiding the variable.

scheme LIST =
class
  variable
    list : **Int***
end

```
public class LIST {
    public static RSLListDefault<Integer> list;
}
```

Example 2.6 Translation of a scheme hiding the variable.

scheme ENCAPSULATED_LIST =
hide list **in**
class
  variable
    list : **Int***
end

```
public class ENCAPSULATED_LIST {
    protected static RSLListDefault<Integer> list;
}
```

Renaming

The purpose of the renaming in RSL is to rename an identifier in relation to other modules. The form of a renaming class expression is:

use id_1 **for** $\mathrm{id}_2, \ldots, \mathrm{id}_m$ **for** id_n **in** class_expression

In Java, there is no concept of renaming identifiers, therefore the renaming must be done before the specification is translated. In RSL a renaming class expression can be expanded into a basic class expression simply by renaming all the identifiers in the class expression according to the list of renaming pairs as shown in Example 2.7. In the translation into Java this expansion should be used.

Example 2.7 Renaming class expression.

scheme STACK_SCHEME =
use stack **for** list **in**
LIST_STATE

Is equivalent to:

scheme STACK_SCHEME =
class
  variable
    stack : **Int***
end

2.2.4 Parameterized Schemes

This project does not define an exact translation for parameterized schemes in RSL. This section discusses some of the issues in the definition of a translation for parameterized schemes. The discussion is included to ensure that the translation of objects and classes defined in this work does not present problems in later extensions of the project.

RSL allows for a wide variety of parameterization of schemes defined in a specification. A scheme is parameterized with objects. One possible use of this is a scheme defining a list, in which the type of the elements of the list has been moved to a parameter of the scheme. The scheme is shown in Example 2.8.

This concept of defining a general class, which can later be instantiated with a specific type is also known in programming languages. In C++ it is known as templates. In Java, the concept has recently been added in version 1.5 under the name generics. The parameterized scheme presented

Example 2.8 Parameterized list in RSL.

```
scheme PARAM_LIST(E: class type Elem end) =
class
   variable
      list : E.Elem*
      ⋮
end
```

Example 2.9 Translation of a parameterized list in Java.

```
public class PARAM_LIST<E> {

    public RSLList<E> list;

    ...
}
```

in Example 2.8 could in Java be translated to the generic class shown in Example 2.9.

Generic methods must be dynamic in Java. The reason for this is that at the time of invocation of the method, the generic type must be known. In Java, the only way to declare the type of a generic class is to instantiate an object of the class with the generic type as type–parameter. A static method in Java does not require an instantiation of an object and the generic type can therefore not be determined. This is the reason that only dynamic methods are allowed to be generic.

Parameterization in RSL may be done with other kinds of declarations than type declarations since the parameterization is based on objects, which may be instances of class expressions containing any kind of declarations. The scheme parameterization in RSL is a way of writing a more general specification, when a parameterized scheme is used either in an instantiation of an object or in a class expression of another scheme, it must be instantiated with an object. A parameterized scheme instantiated with an object can be expanded to a basic class expression.

This gives two possibilities for translation of an instantiation of a parameterized scheme:

1. Rewrite parameterized schemes as basic class expressions and translate the basic class expressions.

2. Define a number of translations depending on the kind of parameterization.

The first suggestion is obiviously the most simple solution, but it has the drawback that it creates one large class rather than dividing the properties between several entities which is the point of the parameterized scheme. An example of an instantiated parameterized scheme and the equivalent expanded class expression is shown in Example 2.10 and Example 2.11.

Example 2.10 Instantiation of parameterized scheme.

```
scheme SIZE =
class
  value
    size : Nat
end

scheme SIZED_INTLIST(S : SIZE) =
class
  variable
    list : Int*
  value
    empty : Unit → write list Unit
    empty() ≡ list := ⟨⟩,
    add : Int → write list Unit
    add(e) ≡ list := ⟨e⟩ ⌢ list
      pre len list < S.size
end

object S :
class
  value
    size : Nat = 7
end

object USESIZEDLIST : SIZED_INTLIST(S)
```

The second suggestion is much more complex to translate, but also provides a solution which is more in line with the idea of the specification. First of all, this project only considers explicit specifications, therefore the use of

Example 2.11 Expanded instantiated parameterized scheme equivalent.

```
object USESIZEDLISTEXPANDED :
class
   variable
      list : Int*
   value
      empty : Unit → write list Unit
      empty() ≡ list := ⟨⟩,
      add : Int → write list Unit
      add(e) ≡ list := ⟨e⟩ ⌢ list
         pre len list < 7
end
```

parameterization to combine axioms are not considered. Only parameterization with variables, values and types are considered, parameterization with channels are part of specification of parallel systems and is not considered. However, the channel concept has some similarities with channels in Java, which may be placed as a field in a class, a discussion of possible translation of channels can be found in Section 7.3.2. Therefore, the translation of parameterization with variables and values could also be considered for channels.

As shown later, type definitions are translated as one or more classes, the translation of a parameterized scheme using types as a parameter could therefore be translated using generics classes in Java as presented in Example 2.9.

Parameterization with variables and values presents some other challenges. Example 2.10 shows a scheme SIZED_INTLIST which is a specification of a list of limited size. The size of the list is put as a parameter to the scheme to allow different sizes of lists. The challenge of the translation of these schemes and objects is to ensure that it is possible to create lists of different sizes which may be referenced from other modules. It should noted that a value in RSL is constant, i.e., it cannot be changed, therefore, the translation should ensure that this is also the case. The translation of a value in the parameter of a scheme could be translated as an argument to the constructor of the class created and kept in the class as a private finalized field to ensure that it cannot be changed. The same translation could be used for variables except that the field in Java should not be declared **final** so that methods could manipulate the translation of the variable. Objects

used in the instantiation of a parameterized are not translated as objects are elsewhere. An example of the translation of the parameterized scheme in Example 2.10 is presented in Example 2.12.

Example 2.12 Translation of parameterized scheme presented in Example 2.10.

```
public class SIZED_INTLIST {
    private final int size;
    public RSLList<Integer> list;

    public SIZED_INTLIST(int size) {
        this.size = size;
    }

    public void empty() {
        list = new RSLListDefault<Integer>();
    }

    public void add(int e) {
        assert list.len() < size;
        list = (new RSLListDefault<Integer>(e))
            .concat(list);
    }
}
```

2.2.5 Object Arrays

This project does not consider a translation of object arrays in RSL. The purpose of the discussion in this section is to ensure that the selection of translation for schemes and objects does not become prohibitive for later extensions allowing the use of object arrays in a specification.

In RSL, it is possible to group a number of objects of the same type into an object array. The objects do not have to be named themselves, but may be referenced by an index to the array. The concept seems fairly correlated to an array of objects in Java. There are a number of differences between object arrays in RSL and arrays in Java. In RSL, an array may be indexed using any type including **Int**, which can be infinite, making the object array infinite in size. In Java, an array has a maximum size based on the limit of *int*, which is used for indexing. Furthermore, arrays in Java may only be indexed using integers and not other types. Therefore, an object array in RSL cannot be translated directly as an array in Java.

In order to translate an object array in RSL into an array in Java, objects in RSL must be translated as instances of classes in Java. Furthermore, a number of conditions of an object array must be met:

1. The typing of the object array in RSL must be finite.
2. It must be possible to calculate the number of elements in the array.
3. The number of elements must not exceed the total number of elements allowed in an array in Java.

If the conditions presented above are met, then an object array in RSL could be translated as an array in Java. An object array declaration and a possible translation is shown in Example 2.13.

Example 2.13 Translation of an object array declaration.

object
O[f : Finite_Typing] : SCHEME
type
Finite_Typing

Is translated as:

```
SCHEME[] O = new SCHEME[/* Calculated size of Finite_Typing */];
private static int getIndex(Finite_Typing ft) {...}
```

2.2.6 Overview of translation of top–level structures

This section presents an overview of the translations of the top–level structures in RSL. It summarizes the choices made in the previous sections.

Unparameterized Schemes

The form of an unparameterized scheme is:

scheme id = ce

where `ce` is a class expression.
The translation in Java is:

```
class ID { CE }
```

ID is the translation of id
CE is the translation of ce

An unparameterized scheme which is not used in instantiations of objects is translated using static methods to make it possible to create a library which does not need an instantiation of an object in Java.

Basic Class Expressions

The form of a basic class expression is:

class
⋮
Constituent declarations
⋮
end

The translation of a basic class expression is a translation of each of its constituent declarations.

1. Translation of type declarations: see Section 2.3.
2. Translation of value declarations: see Section 2.4.
3. Translation of variables is not covered in this work. Suggestion for a translation is given in Section 7.3.1.
4. Translation of channels is not covered in this work. Suggestion for a translation is given in Section 7.3.2.
5. Translation of axioms is not covered in this work.

Extending Class Expressions

Extending class expressions having the form:

extend ce **with** ce_2

where `ce` and ce_2 are class expressions. ce_2 must be a basic class expression. If `ce` is a scheme instantiation then the translation is:

class ID **extends** CE {/* *Translation of declarations of* ce_2 */}

ID is determined from the context. It is either the translation of the name of the scheme containing the extending class expression or determined from the identifier of the object containing the class expression. If `ce` is not a scheme instantiation then the translation is done as if `ce` had the form:

scheme id = ce

where `id` is an auxiliary identifier created in the translation.
The translation is then the same as if `ce` is a scheme instantiation.

Renaming Class Expressions

Renaming class expressions having the form:

use id_{11} **for** id_{12}, ..., id_{n1} **for** id_{n2} **in** ce

where `ce` is a class expression.
This form can be unfolded to a basic class expression[1] by substituting $\texttt{id}_{i1}$ for $\texttt{id}_{i2}$ for $i \in \{1, \ldots, n\}$ in `ce`.

There are no possibilities for renaming identifiers in Java, therefore, the translation is a translation of the unfolded class expression.

Hiding Class Expressions

Hiding class expression having the form:

hide id-list **in** ce

The idea of the translation into Java is to translate the class expression `ce` and afterwards traverse the translated Java and change the modifiers according to the `id`–list. Hiding of types are not covered in this work. Hiding of values and the suggested translation of variables should be done by replacing the **public** modifier with the **protected** modifier.

Scheme Instantiations

The translation of scheme instantiation should be done by unfolding the class expression and translate that, except when dealing with extending class expressions which was explained above.

[1] This is only possible if there are no hidings.

Parameterized Schemes

A parameterized scheme having the following form:

$s(O : ce) = ce_2$

where `ce` is a basic class expression and only contains type declarations, explicit value definitions and variables.
Translation in Java:

```
S<T_a ,... ,T_z> {
  private [final] V_1 _v1;
  ⋮
  private [final] V_n _vn;
  S(V_1 _v1 ,... , V_n _vn) {
    this._v1 = _v1;
    ⋮
    this._vn = _vn;
  }
  /* Translation of ce_2 */
}
```

S is the translation of the name s.
T_a to T_z is the translation of the names of the type declarations of `ce`.
V_1 to V_n is the translation of the types of the values and variable declarations of `ce`
[final] is used in a translation of value declarations and not in a translation of variable declarations.

Translation of a parameterized scheme is a class, the nature of the class in Java depends on the kind of parameterization. Parameterization with types are translated using the generic feature of Java. Parameterization with values and variables are done by translating them as fields and initialize them in the constructor.

Objects

The translation of objects depends on the nature of the class expression included in the object declaration.
Objects declared with a scheme instantiation:

object
$O : id_s$

Is translated as:

```
Id_s O = new Id_s();
```

Id_s is the translation of id_s.

Objects declared with other class expressions than scheme instantiations:

O : ce

Is translated as:

```
class _Id {/* Translation of ce*/}
_Id O = new _Id();
```

_Id is a created identifier from id.

2.3 Types in RSL

In RSL, there are two kinds of types: simple types and complex types, and additionally there are a number of type definitions.

The types and type definitions are described in the following sections along with the chosen translation in Java and a discussion of alternatives considered. First the simple types are described, then the complex types and finally the type definitions are described.

2.3.1 Built–in Types

The simple types are **Bool**, **Int**, **Nat**, **Real**, **Char**, **Text**, and **Unit**. For all these built–in types two operators for comparing values of the type are defined, namely = and ≠. All these types in RSL must be represented in some way in Java. In Java, there are also a number of built-in types for representing truth values as well as numerical values, characters and texts. Two suggestions for translation of the built–in types are given:

1. Represent the built–in types of RSL, using primitive types and library classes in Java.

2. Implement a collection of classes in Java for representing the built–in types of RSL.

Both of these suggestions have a number of advantages and disadvantages.

Using Primitive Types and Library Classes in Java

The first advantage of this suggestion is that value literals in RSL can be translated as value literals in Java, thereby avoiding having to create an object for each value literal in RSL. The second advantage is that expressions using the standard arithmetic operators can be translated using infix expressions in Java, rather than method invocations. The reason for this is that the built–in operators in Java cannot be overloaded, and therefore the only expressions allowed to use them are those where the operands are of primitive types. Both these advantages makes it easier to recognize the specification in the translation. A third important advantage of using primitive types for representing the built–in types of RSL is an efficiency issue. It is much more efficient to represent a numerical value in Java using a primitive type than an instance of a class.

The main disadvantage of this solution is that the types for representing numerical values in Java are limited in their size, whereas the numerical types of RSL are infinite, therefore some mechanism must be provided for handling the limit of a type in Java.

The reason that the suggestion is a combination of both primitive types and library classes in Java is that the primitive types cannot be used in collections in Java. Therefore, the translation needs to use the wrapper classes of the primitives types, when dealing with collections in Java.

Using a Collection of Classes Implemented

The main advantage of this suggestion is that each of the types in RSL are only represented as one type in Java and not as a combination of a primitive type and a wrapper class. Another advantage is that the problem of representing infinitely large values of RSL can be handled inside the classes implemented. It should be noted that it is not possible to represent values of infinite size or precision on a machine with a limited storage.

The main disadvantage is that it is not possible to overload the standard operators in Java, therefore all infix operators must be translated as methods rather than operators in Java, which makes it harder to recognize a specification in the translation of a specification.

Choice of representation of built–in types of RSL

Both of the presented suggestions have advantages and disadvantages. In this work the first suggestion have been chosen, because it is the most intuitive solution. There are several possible solutions to the problem with the sizes of the types in Java.

1. Let the user decide how to translate each of the built–in types.

2. Let the user decide the translation of each use of a built–in type via meta data in the specification.

3. Use the widest type to reduce the number of times a limit causes troubles.

4. Decide on a type representation in Java, which suits most situations.

The first and third of the four solutions have a problem in case of selecting a large type like *BigInteger* for representing **Int** and then having a for–loop construction with values 1 to 5. In this case the selection of *BigInteger* would be very inefficient, but it may be needed in other parts of a specification. The second solution, where different types may be used, presents a problem because some type casts may be needed when assigning a value of a wider type to a narrower one. Type casting should only be performed with great care of a programmer since the value being type cast may be too large to fit within the new type. The fourth solution was chosen for this project, but with the requirement, that it should be relatively easy to change the type in the tool developed.

Type Bool

The type **Bool** in RSL represents the boolean values, **true** or **false**. In Java, there is a type containing the truth values namely *boolean* or wrapped as an object *Boolean*. The primitive value *boolean* and the wrapper class *Boolean* in Java were chosen to represent the RSL type **Bool**.

Two operators have been defined and a number of connectives for values of the type **Bool**. They are presented in Table 2.1 along with their translation into Java.

RSL	**Translation into Java**
b1 = b2	b1 == b2 or b1.equals(b2)
b1 ≠ b2	b1 != b2 or !b1.equals(b2)
∼ b1	!b1
b1 ∧ b2	b1 && b2
b1 ∨ b2	b1 || b2
b1 ⇒ b2	**if**(b1) b2 **else true**

Table 2.1: Operators and connectives for type **Bool**

The use of == or equals and != or !equals depends on whether the type of the operands involved are of the wrapper class, *Boolean*, or the primitive type *boolean*. == and != are defined for both objects and the primitive types. It is therefore necessary to determine, whether to use the operators or the methods. The operators == and != for objects determine whether or not it is the same object, and not whether or not the value of the object is the same. For the connectives this is not a problem since these are not defined for objects and Java can therefore automatically unwrap the objects before using the operator.

Type Int

The type **Int** in RSL represents the type containing integer values in the range ..., -2, -1, 0, 1, 2, ... In Java, there are several types for representing integer values depending on the range needed. The Java types for representing integer values are shown in Table 2.2 along with their ranges.

Type	Range
`byte`	-128 to 127
`short`	-32,768 to 32,767
`int`	-2,147,483,648 to 2,147,438,647
`long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`BigInteger`	no range [a]

[a] No limit but in practice limited by the size of the memory of the machine.

Table 2.2: Overview of Java types for representing integer values.

This project will use *int* and the wrapper class *Integer* for representing the RSL type **Int**. A number of operators have been defined for values of the RSL type **Int**. These are shown in Table 2.3 along with their translation in Java.

The choice of whether to use == or equals as well as their negated counterparts is the same as for type **Bool**.

The integer division in RSL returns the whole number of times the first operand goes into the second. The same is true for the division operator / in Java. The integer remainder operator \ in RSL handles negative values the same way the Java modulo operator % does, by returning the absolute value of the remainder with the sign of the first operand.

RSL	Translation in Java
i = j	i == j or i.equals(j)
i ≠ j	i != j or !i.equals(j)
i > j	i > j
i ≥ j	i >= j
i < j	i < j
i ≤ j	i <= j
i + j	i + j
i − j	i − j
i ∗ j	i ∗ j
i / j	i / j
i \ j	i % j
i ↑ j	(**int**) Math.pow(i, j)
real i	(**double**)i

Table 2.3: Operators for the type **Int**

Type Nat

The type **Nat** in RSL represents the type containing the natural numbers, which are the same as the values in **Int**, except that **Nat** contains only the positive values and zero. The operators for **Nat** are exactly the same as for **Int**. The **Nat** type in RSL is actually a subtype of **Int**, and the handling of subtypes is discussed in Section 2.3.4.

Type Real

The type **Real** in RSL represents all the real numbers and may have values like . . . , -2,3, 0.0, 5.7, . . . and with an infinite number of decimals i.e. infinite precission. In Java real numbers are represented by a number of types. Each of them are shown in Table 2.4 along with their ranges.

Type	Range
`float`	3.8×10^{-38} to 3.8×10^{38}
`double`	1.7×10^{-308} to 1.7×10^{308}
`BigDecimal`	no range [a]

[a] Based on the BigInteger and has the same limitations.

Table 2.4: Overview of Java types for representing real values.

The primitive type *double* and the wrapper class *Double* will be used to represent the RSL type **Real** in this project. The operators on values of the

type **Real** are the same as for **Int**. The only exception is that the ↑ returns a **Real** instead of an **Int** and the translation is therefore Math.pow(i, j). The standard arithmetic operators and relational operators in Java are overloaded and also works for values of type *double*, therefore the translations are the same as for **Int**. The conversion operator **real** is only defined for type **Int**, and therefore a translation is not defined. The last built–in operator defined for **Real** is an operator for converting a **Real** to an **Int**. This operator is called **int** and like the operator **real** it is translated as type casting: (**int**) r where r is of type double. Both the RSL operator **int** and type casting (**int**) returns the integer closest towards zero, i.e. truncation.

Type Char

The type **Char** represents individual characters in RSL. In Java, there is also a type for representing single characters namely *char*. Therefore, the translation of type **Char** in RSL is *char* and *Character* in Java. The translation of the two operators for equality and inequality is the same as for the other built–in types described. No other operators have been defined for the RSL type **Char**. The character set allowed in RSL contains the English alphabet, digits and a number of graphics characters. The graphical characters in RSL are not represented in the same way as in Java, therefore a translation between the two character sets is needed. The treatment of special characters in RSL are not considered in this work, but it would only present a minor extension.

Type Text

The type **Text** in RSL represents strings of characters. In Java, there is also a type for representing strings of characters and that is the class *String*. *String* is therefore chosen as translation of type **Text**. In RSL, **Text** is defined as a list of characters and therefore the standard list operators are also valid for values of type **Text**. This leads to the idea of translating **Text** as a list of characters in Java. This would, however, lead to a very inefficient solution since each character would have to be represented by an object, and the possibility of using the standard notation for *String* in Java would be lost. The standard list operators do not have a direct translation in Java. They are translated using the methods in the class *String*.

The standard list operators for **Text** are translated as shown in Table 2.5.

RSL	Translation in Java
⌢	String concat(String str)
hd	**char** charAt(0)
tl	String substring(1)
len	**int** length()
elems	**new** RSLSetDefault<Character>(s.toCharArray())[a]
inds	**new** RSLSetDefault<Integer>.range(1, s.length())[a]

[a] For a description of RSLSetDefault please see Section 2.3.2.

Table 2.5: Overview of translations of **Text** operations in Java

Type Unit

The type **Unit** in RSL represent a single empty value. In RSL, it is often used in the type expression of a function for either representing, that the function has no return value, or that it does not take any arguments. In Java, a method with no return value uses the keyword *void*. A method in Java that does not have any arguments is just written with empty parentheses after the method identifier. In RSL the two operators for equality and inequality also works for **Unit**. The use of the equality operator on two expressions evaluating to **Unit** is true, while use of the inequality operator is false. To obtain the same properties in Java, a special type for representing a single value should be implemented, since values of type void are not allowed in Java.

Translations of the built–in RSL types have been summarized in Table 2.6.

RSL	Java
Bool	**boolean** or Boolean
Int	**int** or Integer
Nat	No translation.
Real	**double** or Double
Char	**char** or Character
Text	String
Unit	No translation. Functions should use the keyword **void** and ().

Table 2.6: Summary of translation of built–in types in RSL.

2.3.2 Complex Types

There are several complex types in RSL, namely: product, function, list, map and set. Common for the complex types are that they involve other types, which may be: Simple types, complex types, or names of type definitions. Lists, sets and maps in RSL may both be finite and infinite, but due to the limitations of Java, this work only considers the finite versions. Structures of infinite size are not well suited for an executable program. However, it should be noted, that there are a number of special cases in which infinite lists, sets and maps can be evaluated, these special cases are not considered in this project. The three complex types list, set and map have direct counterparts in Java in form of a number of interfaces extending the general *collection* interface. The two complex types, product and function, do not have any direct counterparts in Java.

Product Types

An RSL product is defined as a finite ordered collection of possibly different types [11]. Product type expression are of the form:

$\text{type\_expr}_1 \times \ldots \times \text{type\_expr}_n$

Values of a product type of this form is a tuple of length n:

$(\text{value\_expr}_1, \ldots, \text{value\_expr}_n)$

value_expr$_i$ is a value of type *type_expr*$_i$.

The only way in Java to create a new type is to create a class. Therefore a product type expression in RSL is translated as a class in Java. Each *type_expr*$_i$ is translated as a public field in the class with an identifier "_v[i]" and the type of the field is the translation of the *type_expr*$_i$.

There are no operations defined on products except for $=$ and $\neq$ which should be translated as p_1.equals(p_2) and !p_1.equals(p_2). The equals method must call the equals method for all subparts of reference types and use an infix expression with the == operator for all subparts of primitive types. An example of the translation of a product type is shown in Example 2.14.

Function Types

In RSL, a function is a value like any other value. This means that functions may be used as parameters to other functions. It is possible to specify higher order functions as well as to compose functions from other functions. The

Example 2.14 Translation of a product.

SYSTEM_OF_COORDINATES =
class
  value
    origo : **Unit** → **Real** × **Real**
    origo() ≡ (0.0, 0.0)
end

```
public class _C {
    public double _v1;
    public double _v2;

    public _C(double _v1, double _v2) {
        this._v1 = _v1;
        this._v2 = _v2;
    }

    public boolean equals(Object o) {
        if(o instanceof _C) {
            return ((this._v1 == o._v1) &&
                    (this._v2 == o._v2));
        }
        return false;
    }
}

public class SYSTEM_OF_COORDINATES {

    public static _C origo() {
        return new _C(0.0, 0.0);
    }

}
```

idea of using a function as a parameter to another function has no direct counterpart in Java. In Java, methods represent a special type, and it is not possible to create values of that type. There exist a class *Method* in Java, and an instance of the class *Class* contains a collection of these methods. However, it is not possible to modify the collection of methods and it is not possible to create an instance of the class *Method*.

RSL provides two constructors, $\rightarrow$ and $\xrightarrow{\sim}$, for building function types. Only the former constructor is considered, it constructs total function, whereas the latter constructs partial functions, a kind of under specification which is not considered in this project.

In C or C++ the use of functions as parameters could be solved using pointers to functions, but this is not possible in Java. One possible solution is to use an interface containing a method called action, and let different values of function types in RSL be represented as an object implementing the interface. An example of the possible implementation in Java is shown in Example 2.15

The function expression, which is taken as parameter to the function `twice`, is translated as an interface containing one method action which has the same signature as the function type expression of the parameter. The first test case presented in the example is not very useful, but the result of calling `twice` with one parameter could be used as input for another higher order function taking a function from **Int** to **Int** as parameter.

The translation of an uncurried version of the function `twice` could be translated using a similar scheme, which is shown in Example 2.16.

List Types

A list in RSL is a sequence of values of the same type [11]. The concept of a list can also be found in Java. A list in Java is represented by the interface *List*. The interface *List* in Java defines a number of methods for manipulating a list. When comparing the methods in the *List* interface and the operators on lists in RSL, one can see, that most of the operators on lists exists as methods in the Java interface even though the names and exact use are a little different.

An interface, *RSLList*, has been implemented along with an implementing class, *RSLListDefault*. The source code for *RSLList* can be found in Appendix E.2.1 and the source code for *RSLListDefault* can be found in Appendix E.2.2. The combination of an interface and an implementing class allows others to develop a different implementation of a representation for lists.

The interface defines methods, which correspond directly to the operators

Example 2.15 Defining a function using an interface.

value
  twice : (**Int** → **Int**) → (**Int** → **Int**)
  twice(f, i) ≡ f(f(i))
test_case
  [t1] twice(λ i : **Int** • i + 1),
  [t2] twice(λ i : **Int** • i − 1)(1)

```
interface IntXInt {
    public int action(int _v1);
}

public static IntXInt twice(final IntXInt f) {
    return new IntXInt() {public int action(int i){
        return f.action(f.action(i));}};
}

public static void main(String[] args) {
    System.out.println("[t1]: " +
        twice(new IntXInt() {
                public int action(int i) {return i + 1;}}));
    System.out.println("[t2]: " +
        twice(new IntXInt() {
                public int action(int i) {return i - 1;}}).
                    action(1));
}
```

Example 2.16 Defining an uncurried function using an interface.

value
twice : (**Int** → **Int**) × **Int** → **Int**
twice(f, i) ≡ f(f(i))
test_case
[t1] twice(λ i : **Int** • i + 2, 3)

```
interface IntXInt {      public int action(int _v1); }

public static int twice(final IntXInt f, int i) {
    return f.action(f.action(i));
}

public static void main(String[] args) {
    System.out.println("[t1]: " +
       twice(new IntXInt() {
                public int action(int i) {return i + 2;}},
             3));
}
```

on lists in RSL. The implementing class is based on a collection implementing the *List* interface. New implementations of the interface must at least contain the same constructors as *RSLListDefault*. The constructors are not specified in the interface, because this is not allowed in Java.

The translation from RSL into Java is as follows:
RSL:
$t^\star$
Java:
RSLListDefault<T>, where T is the translation of t[2].

As stated previously infinite lists have been left out. In RSL, a new list can be created in three ways; either by enumerating the elements, specifying an interval, or by a comprehended list expression. The translation of the two first methods for creating lists is straightforward. *RSLListDefault* contains a constructor taking an array which holds the enumerated elements, and a constructor taking two integers for specifying an interval. The comprehended list expression consists of three parts: a list, an expression which must be

[2]The notation <T> is standard in Java 1.5 for the generic type of the class. Please see Section 4.1 on page 105 for an explanation of the new generic feature of Java 1.5.

applied for each element in the list before inserting it into the new list, and an optional condition which an element from the old list must fulfill, in order to be inserted into the new list. This form of list creation is implemented as a method *listComp* in the RSLList interface taking two parameters: an *RSLExpression* and an optional *Testable*. *RSLExpression* represents the expression, which must be applied for each element before it is inserted into the new list. *RSLExpression* is an generic interface and in order to use it, one must create a class implementing the generic method *action*. *Testable* represents the condition which must be fulfilled for each element to be inserted into the new list. *Testable*, like *RSLExpresion*, is a generic interface and contains a method *test*. The translation of the three forms of list creation is defined below.

RSL:

$\langle$value_expr$_1$, ..., value_expr$_n\rangle$
$\langle$value_expr$_i$ .. value_expr$_j\rangle$
$\langle$value_expr$_e$ | binding **in** value_expr$_l$ • value_expr$_c\rangle$

Java:

```
new RSLListDefault<T1>(new T1[]{V1 , ... , Vn})
new RSLListDefault<Integer>(Vi , Vj)
Vl.listComp(
  new RSLExpression<Te>() {public Te action(Te b) {return (Ve);}},
  new Testable<Te>(){public boolean test(Te b) {return (Vc);}}
)
```

T_1 is the translation of the type of the elements determined from the type of the first element value_expr$_1$.
V_1 to V_n are the translations of the value expressions value_expr$_1$ to value_expr$_n$.
V_i and V_j are the translations of the value expressions value_expr$_i$ and value_expr$_j$.
V_l is the translation of value_expr$_l$ which must evaluate to a list.
T_e is the translation of the type of value_expr$_e$.
b is the translation of the binding
V_e is the translation of value_expr$_e$ and V_c is the translation of value_expr$_c$.

Examples of all three kinds of list creation and their translation in Java are shown in Example 2.17.

In RSL, a number of operators on lists have been defined. These operators are implemented as methods in the *RSLList* interface. As an example

Example 2.17 List creation – examples.

⟨**true**, **false**, **true**, **false**⟩,
⟨1..10⟩
⟨2∗n | n **in** ⟨0,1,2,3⟩ • n \ 2 = 0⟩

```
new RSLListDefault<Boolean>(new Boolean[]{true, false, true,
    false});
new RSLListDefault(1, 10);
(new RSLListDefault<Integer>(new Integer[]{0,1,2,3})).
    listComp(
       new RSLExpression<Integer>(){
                public Integer action(Integer n) {return new
                    Integer(2*n); }},
            new Testable<Integer>(){
                public boolean test(Integer n) { return (n
                    % 2 == 0); }}
    );
```

concatenation of lists denoted by ⌢ is translated into the method RSLList< E> concat(RSLList<E>). The translation of the operators are shown in Table 2.7.

Operator	Translation
⌢	RSLList<E> concat(RSLList<E>)
hd	E hd()
tl	RSLList<E> tl()
len	**int** len()
elems	RSLSet elems()
inds	RSLList<Integer> inds()
(i)	<E> get(i)
(i) = j	**void** set(i, j)

Table 2.7: Overview of translation of list operators in RSL to Java.

The use of the operators and translation of their use are shown in Example 2.18.

Set Types

The complex type set in RSL is an unordered collection of distinct values of the same type [11]. The set type in RSL consists of both finite and infinite

Example 2.18 List operators – examples.

⟨1,2,3,4,5⟩ ⌢ ⟨6,7,8,9,10⟩
⟨1,2,3,4,5⟩(2)
hd ⟨1,2,3,4,5⟩
tl ⟨1,2,3,4,5⟩
len ⟨1,2,3,4,5⟩
elems ⟨2,2,2,4,4,4⟩
inds ⟨2,2,2,4,4,4⟩
⟨5,4,3,2,1⟩(1)
il(i) = 7

```
(new RSLList(new Integer[]{1,2,3,4,5})).concat(
   new RSLList(new Integer[]{6,7,8,9,10}));
(new RSLListDefault(new Integer[]{1,2,3,4,5})).get(2);
(new RSLListDefault(new Integer[]{1,2,3,4,5})).hd();
(new RSLListDefault(new Integer[]{1,2,3,4,5})).tl();
(new RSLListDefault(new Integer[]{1,2,3,4,5})).len();
(new RSLListDefault(new Integer[]{2,2,2,4,4,4)).elems();
(new RSLListDefault(new Integer[]{2,2,2,4,4,4)).inds();
(new RSLListDefault(new Integer[]{5,4,3,2,1})).get(1);
il.set(i, 7);
```

sets. Only finite sets are considered here. In Java, there exists an interface *Set*, which defines the concept of a set in Java. An instance of a class implementing the *Set* interface holds the same properties as a set in RSL. As for the translation of lists in RSL an interface, *RSLSet*, defining methods corresponding to the operators in RSL has been defined. An implementing class, *RSLSetDefault*, based on a variable of a class implementing the *Set* interface has also been defined.

The translation from RSL into Java is as follows:
RSL:
t-**set**
Java:
RSLSetDefault<T>, where T is the translation of t.

In RSL, sets may be created in three ways; by enumerating the elements, as an interval of integers, or by a comprehended set expression. Comprehended set expressions are not supported, because they can result in infinite sets. However, if the comprehended set expression is on a special form it is possible to define a translation:

$\{F(x)|x:t\bullet\ x \in S \wedge P(x)\}$

S must be a finite set and both the predicate P and the expression F must be possible to translate using the same idea as for lists.

The enumerated and ranged set expressions are translated like the enumerated list expressions and ranged list expressions. The translation from RSL into Java of these two kinds of set expressions are defined below:

RSL:

$\{value\_expr_1, \ldots, value\_expr_n\}$
$\{value\_expr_i\ ..\ value\_expr_j\}$

Java:

```
new RSLSetDefault<T1>(new T1[]{V1,...,Vn })
new RSLSetDefault<Integer>(Vi, Vj)
```

T_1 is the translation of the type of the elements determined from the first element.
V_1 to V_n is the translation of the value expressions $value\_expr_1$ to $value\_expr_n$.
V_i and V_j is the translation of the two value expressions $value\_expr_i$ and

value_expr$_j$.

Examples of the two ways of creating set are shown in Example 2.19.

Example 2.19 Set creation - examples.

{1,2,3,4}
{1..4}

```
new RSLSetDefault<Integer>(new Integer[]{1,2,3,4});
new RSLSetDefault<Integer>(1,5);
```

In RSL, a number of operators on sets have been defined. The translation of these operators are defined as methods in the *RSLSet* interface and implemented in the *RSLSetDefault* class. The translation of the operators are shown in Table 2.8

Operator	Translation
$\in$	**boolean** isIn()
$\ni$	**boolean** isNotIn()
$\cup$	RSLSet<E> union(RSLSet<E>)
$\cap$	RSLSet<E> intersect(RSLSet<E>
$\setminus$	RSLSet<E> difference(RSLSet<E>)
$\subseteq$	**boolean** subSet(RSLSet<E>)
$\subset$	**boolean** properSubSet(RSLSet<E>)
$\supseteq$	**boolean** superSet(RSLSet>E>)
$\supset$	**boolean** properSuperSet(RSLSet<E>)
card	**int** card()

Table 2.8: Overview of translation of set operators in RSL to Java.

The use of the operators are shown in Example 2.20 as well as their translation in Java.

Map Types

The complex type map in RSL is a table–like structure, which maps values of one type into values of possibly another type [11]. Maps may be infinitely large. Like for lists and sets, only finite versions of maps are considered in this project. In Java, there is an interface named *Map*, which serves the

Example 2.20 Use of infix operators on sets.

$1 \in \{1,2,3,4,5\}$
$7 \notin \{1,2,3,4,5\}$
$\{1,2\} \cup \{3,4\}$
$\{1,2\} \cap \{2,3\}$
$\{1,2,3,4,5\} \setminus \{4,5\}$
$\{1,2,3\} \subset \{1,2,3\}$
$\{1,2,3\} \subseteq \{1,2,3\}$
$\{1,2,3\} \supset \{1,2,3\}$
$\{1,2,3\} \supseteq \{1,2,3\}$
card $\{1,2,3,4\}$

```
(new RSLSetDefault<Integer>(new Integer[]{1,5})).isin(1);
(new RSLSetDefault<Integer>(new Integer[]{1,5})).isNotin(7);
(new RSLSetDefault<Integer>(new Integer[]{1,2})).union(
     new RSLSetDefault<Integer>(new Integer[]{3,4}));
(new RSLSetDefault<Integer>(new Integer[]{1,2})).intersect(
     new RSLSetDefault<Integer>(new Integer[]{2,3}));
(new RSLSetDefault<Integer>(new Integer[]{1,2,3,4,5})).
   difference(
     new RSLSetDefault<Integer>(new Integer[]{4,5}));
(new RSLSetDefault<Integer>(new Integer[]{1,2,3})).subSet(
     new RSLSetDefault<Integer>(new Integer[]{1,2,3}));
(new RSLSetDefault<Integer>(new Integer[]{1,2,3})).properSubSet(
     new RSLSetDefault<Integer>(new Integer[]{1,2,3}));
(new RSLSetDefault<Integer>(new Integer[]{1,2,3})).superSet(
     new RSLSetDefault<Integer>(new Integer[]{1,2,3}));
(new RSLSetDefault<Integer>(new Integer[]{1,2,3})).
   properSuperSet(
     new RSLSetDefault<Integer>(new Integer[]{1,2,3}));
(new RSLSetDefault<Integer>(new Integer[]{1,2,3,4,5})).card()
```

same purpose namely to provide a mapping of keys into values. In RSL, it is possible to have a non-deterministic map, i.e., a map where a value of the domain is mapped to several values in the range of the map. Since non-determinism is seldom wanted in an implemented system, non–deterministic maps are not considered here.

The implementation of maps in the translation are very similar to the implementation of lists and sets. An interface and an implementing class have been developed. The interface is named *RSLMap* and the implementing class *RSLMapDefault*. The methods defined in the interface correspond closely to the operators defined on maps in RSL.

The translation from RSL into Java is as follows:
RSL: $t_1 \ \overrightarrow{m} \ t_2$
Java: RSLMapDefault<T_1 , T_2>, where T_1 is the translation of t_1 and T_2 is the translation of t_2.

In RSL, there are two ways to create a map; either by enumerating the elements as pairs or by creating a comprehended map expression. The comprehended map expression consist of two value expressions, one value expression expressing the domain of the map, and one value expression expressing the range of the map. The comprehended map expression may result in both non–deterministic and infinite maps, and they are therefore disallowed. However, if the comprehended map expression is on a special form it may be translated anyway:

$[\, x \mapsto F(x) | x{:}t \bullet x \in S \wedge P(x) \,]$

S must be a finite set and F(x) and P(x) must be possible to translate as previously done with lists and sets.

The only allowed form of creating maps are by using an enumerated map expression. The definition of the translation from RSL into Java is shown below.

RSL:

$[\, \text{value\_expr}_{d1} \mapsto \text{value\_expr}_{r1}, \ldots, \text{value\_expr}_{dn} \mapsto \text{value\_expr}_{rn} \,]$

Java:

```
new RSLMapDefault<T1 , T2>(
  new T1 [ ] { Vd1 , ... , Vdn } ,
  new T2 [ ] { Vr1 , ... , Vrn }
)
```

T_1 is the translation of the type of value_expr$_{d1}$
T_2 is the translation of the type value_expr$_{r1}$.
V_{ai} is the translation of the corresponding value_expr$_{ai}$.

Examples of the creation of maps are shown in Example 2.21. There are three constructors to allow for different ways of creating maps. The first constructor takes no arguments and creates an empty map. The second constructor takes two objects as argument and creates a map with one entry. The third constructors takes two arrays as arguments and it creates a map where the elements of the first array is mapped into the elements of the second array.

Example 2.21 Creation of maps.

[ ]
[1 ↦ ″one″]
[2 ↦ ″two″, 3 ↦ ″three″]

```
RSLMapDefault<Integer , String>();
RSLMapDefault<Integer , String>(1, "one");
RSLMapDefault<Integer , String>(new Integer[]{2,3}, new String[]{
    "two" , "three"});
```

There are a number of operators in RSL for manipulating maps. These operators consist of both prefix and infix operators, all of which are translated as dynamic methods in Java. The prefix operators are translated as methods taking no arguments, while the infix operators are translated as methods taking one argument. Even though the methods are dynamic and therefore must be invoked on an object they do not change the object, on which they are invoked, instead they return a new object. An example of this is an invocation of the method override on one map with another map as argument, which neither changes the map on which override is invoked, nor the map given as arguments. overwrite returns a new map created from the content of the map on which it is invoked and the map provided as argument. The translation of the operators are defined in Table 2.9.

Operator	Translation
$\cup$	RSLMap<K,V> union(RSLMap<K,V>)
$\dagger$	RSLMap<K,V> override(RSLMap<K,V>)
$\backslash$	RSLMap<K,V> restrictBy(RSLSet<K>)
/	RSLMap<K,V> restrictTo(RSLSet<K>)
$\circ$	RSLMap compose(RSLMap)[a]
dom	RSLSet<K> dom()
rng	RSLSet<V> rng()

[a] The missing generic types is due to the involvement of three types: the key of the inner map, the value of the inner map which is the same as the key of the outer map, and the value of the outer map. Use of more generic types in a method than defined in the class signature is not permitted in Java 1.5.

Table 2.9: Overview of translation of RSL map operators into Java.

The use of the operators and their translations are shown in Example 2.22

2.3.3 Type Definitions

There are a number of type definitions in RSL, which are used to create new types. The type definitions in RSL are:

1. Sort definitions
2. Variant definitions
3. Short record definitions
4. Union definitions
5. Abbreviation definitions

The purpose of type definitions is to create new types, which may be used elsewhere in a specification. The variant definition, e.g. creates a type with several alternatives, which a value of the type may take. In Java, there is only one way to create new types, and that is by creating a new class. Therefore, the translations of type definitions all involve generation of one or more classes in Java.

In RSL, there is structural equivalence between types but not name equivalence. Therefore, the specification in Example 2.23 is valid, even though a variable of one abbreviation type is a assigned to a variable of another abbreviation type. The two abbreviation definitions are abbreviations for the same type, and the assignment is therefore allowed. In Java, on the other hand, there is name equivalence between types. Name equivalence means that two

Example 2.22 Use of map operators.

[1↦1.0,2↦2.0] ∪ [3↦3.0, 4↦4.0]
[1↦1.0,2↦2.0] † [1↦3.0]
[1↦1.0,2↦2.0] \ {2}
[1↦1.0,2↦2.0] / {1}
[1↦1.0,2↦2.0] ° [1.0↦``one″]
dom [1↦1.0,2↦2.0]
rng [1↦1.0,2↦2.0]

```
new RSLMapDefault<Integer , Double>(new Integer[]{1,2},
                                    new Double[]{1.0,2.0}).union(
    new RSLMapDefault<Integer , Double>(new Integer[]{3,4},
                                        new Double[]{3.0,4.0});
new RSLMapDefault<Integer , Double>(new Integer[]{1,2},
                                    new Double[]{1.0,2.0}).
                                        override(
    new RSLMapDefault<Integer , Double>(new Integer[]{1},
                                        new Double[]{3.0});
new RSLMapDefault<Integer , Double>(new Integer[]{1,2},
                                    new Double[]{1.0,2.0}).
restrictionBy(new RSLSetDefault<Integer>(2);
new RSLMapDefault<Integer , Double>(new Integer[]{1,2},
                                    new Double[]{1.0,2.0}).
restrictionTo(new RSLSetDefault<Integer>(1);
new RSLMapDefault<Integer , Double>(new Integer[]{1,2},
                                    new Double[]{1.0,2.0}).
                                        compose(
    new RSLMapDefault<Double , String>(1.0,"one");
new RSLMapDefault<Integer , Double>(new Integer[]{1,2},
                                    new Double[]{1.0,2.0}).dom();
new RSLMapDefault<Integer , Double>(new Integer[]{1,2},
                                    new Double[]{1.0,2.0}).rng();
```

types are considered to be equivalent if the names of the types are the same. In Java, it is possible to create new types containing the same components, and then name them differently. However, it is not possible to assign a value of one of these types to a variable of another type. The two types in Java are different types, whereas in RSL, the two types are the same. There are two possible strategies for solving this.

Example 2.23 Structural equivalence in RSL

```
scheme PRODUCT_TEST =
class
  type
    Product_1 = Int × Real,
    Product_2 = Int × Real
  variable
    p_1 : Product_1 := (1, 1.0),
    p_2 : Product_2 := (2, 2.0)
  value
    test : Unit → read p_1 write p_2 Unit
    test() ≡
      p_2 := p_1
end
```

The first strategy is to invent a naming convention and use this for all abbreviation types defined in RSL, thereby ensuring that all equivalent abbreviations are translated as one type in Java.

The second strategy is to let both abbreviation types be created with the names given, and then ensure by other means that it is possible to assign a variable of one type to a variable of another type. One way to ensure this is to create a common super type and then let variables which are defined be of the super type, rather than one of the specific types.

Both strategies have the drawback, that the translation is quite different from the specification. The first solution is chosen since it provides the code with the least number of classes. The tool carrying out the translation should create a class named with an underscore and the content of the type expression seperated by underscore. The Product_1 type abbreviation in Example 2.23 should result in a Java class: `_Int_Real`.

Sort Definition

A sort definition specifies a type which has yet to be defined. The sort type can be used in type expressions, e.g. in other type definitions. The translation of the sort type should have the same properties. The translation of the sort definition is an empty class, because this resembles the same properties.

The translation from RSL into Java is as follows:
RSL:

type t

Java:

```
public class T {}
```

Where T is the translation of the name t.

Variant Definitions

A variant definition is a convenient way to group a number of values and functions on sort definitions [11]. A variant definition is a grouping of alternatives, which may either be constants or record constructors with components defining the content of the alternative. The types may be recursively defined.

In the translation into Java, the same properties are desired. There is a need for a type which is a grouping of alternatives. There is only one concept in Java which allows for values of different types to be assigned to the same variable, and that is if the types in Java are subtypes of a general type, and that the variable is of the general type.

A variant definition is translated as an abstract class with one extending class for each alternative.

The translation from RSL into Java is as follows:
RSL:

type t_1 == a_1 | ... | a_n

Java:

```
public abstract class T1 {
  public abstract boolean equals(Object o);
  public abstract String toString();
}
```

public class A_1 **extends** T_1 {...}

public class A_n **extends** T_1 {...}

T_1 is the translation of the name t_1.
A_1 is the translation of the name a_1.
A_n is the translation of the name a_n.

Translation of a constant alternative:
RSL:

type t == a

Java:

```
public class A extends T
  public boolean equals(Object o) {
    if(o instanceof A)
      return true;
    else
      return false;
  }

  public String toString() {
    return "A";
  }
}
```

T is the translation of the name t.
A is the translation of the name a.

The reason for the special implementation of the equals method in the class A, is that whenever an A object is created, it should be identical to other objects of the type. This could also be achieved by implementing the singleton design pattern [7]. This solution, however, would lead to two different ways of creating objects, because objects are created by methods and not constructors in the singleton design pattern.
Translation of a record constructor alternative:
RSL:

type t = a(t_2)

Java:

```
public class A extends T {
  public T2 _v1;
```

```
  public A (T2 _v1) {
    this._v1 = _v1;
  }

  public boolean equals(Object o) {
    if(o instanceof A) {
      if(!_v1.equals(((T2) o)._v1) {
        return false;
      }
      return true;
    }
    return false;
  }

  public String toString() {
    return "A(" + _v1 + ")";
  }
}
```

T is the translation of the name t.
A is the translation of the name a.
T_2 is the translation of type t_2.

A record alternative of a variant definition may specify destructors for each of the components it incoorporates. The purpose of a destructor is to be able to retrieve a component of an alternative. A destructor in RSL corresponds very much to a *get* method in Java, which returns the value of a field in a class. An alternative of a variant type definition may also specify reconstructors. The purpose of a reconstructor is to change a part of a record without changing the rest of the record. In Java, the idea of changing a field in a class would normally be done by a *set* method. However, it should be noted that a reconstructor in RSL is just a convenient way for creating a new value, in which one of the components is changed. The translation cannot be a *set* method in Java but must be a method creating a new object.

The translation of destructors and reconstructors in RSL into Java are as follows:
RSL:

type t_1 == a(get_t : $t_2 \leftrightarrow$ replace_t),

Java:

```
public abstract class T1 { //same as earlier defined }

public class A extends T1 {
```

```
  private T2 _v1;

  public A(T2 _v1) {
    this._v1 = _v1;
  }

  public T2 get_t() {
    return _v1;
  }

  public T1 replace_t() {
    return new A(_v1);
  }

  /*toString and equals method omitted follows same pattern as
      earlier defined.*/
}
```

T_1 is the translation of the name t_1.
A is the translation of the name a_1.
T_2 is the translation of the type t_2.

In Example 2.24, a variant definition with two alternatives a constant and a recursively defined record constructor is shown.

If the variant definition consists of only constants, it is tempting just to use an enumeration, as shown in Example 2.25. This would also be more effecient than creating an object for each value of the variant. However, in order to make the translation consistent, the same method is chosen for all kinds of variant definitions.

Example 2.26 shows a variant definitioncontaining destructors, reconstructors, and the translation of these.

The simple wildcard, which does not contain any components, is just ignored, because it is impossible to limit the number of classes extending a general super class in Java. Either a class allows for extension, which is needed because the other alternatives already extend the abstract class or a class can be declared *final* and thereby disallow any extensions. The wildcard constructor, which contains one or more components, is also just ignored.

Variant definitions are not just used for recursive definitions. They can also be used to form a union of types which are otherwise unrelated, as shown in Example 2.27

Example 2.24 Recursive variant type definition

type
Collection == empty | add(Elem, Collection),
Elem

```
public abstract class Collection {
/*equals and toString methods omitted*/
}

public class empty extends Collection {
    public boolean equals(Object o) {
        if(o instanceof empty)
            return true;
        else
            return false;
    };

    public String toString() {
        return "empty";
    }
}

public class add extends Collection {
    private Elem _v1;
    private Collection _v2;

    public add(Elem _v1, Collection _v2) {
        this._v1 = _v1;
        this._v2 = _v2;
    }

    public boolean equals(Object o) {
        if(o instanceof empty) {
            if(!(this._v1.equals(((add) o)._v1))) {
                return false;
            }
            if(!(this._v2.equals(((add) o)._v1))) {
                return false;
            }
            return true;
        }
        return false;
    }
    /*toString method omitted*/
}

/*Translation of Elem omitted*/
```

Example 2.25 Translation of variant type definition using an enumeration.

type
Color == black | white

```
public enum Color{black, white};
```

Example 2.26 Variant type definition having constructors, destructors and reconstructors

type
List == empty | add(head : Elem ↔ replace_head, tail : List)

```
public abstract class List {
    /*equals and toString method omitted*/
}

public class empty extends List {
    /*equals and toString method omitted*/
}

public class add extends List {
    private Elem _v1;
    private List _v2;

    public add(Elem head, List tail) {
        _v1 = head;
        _v2 = tail;
    }

    public Elem head() {
        return _v1;
    }

    public List tail() {
        return _v2;
    }

    public List replace_head(Elem e) {
        return new add(e, _v2);
    }

    /*equals and toString method omitted*/
}
```

Example 2.27 Not recursive variant definition.

type
  Figure ==
    box(length : **Real**, width : **Real**) |
    circle(radius : **Real**) |
    triangle(base_line : **Real**, left_angle : **Real**, right_angle : **Real**)

```
public abstract class Figure {
    /*equals and toString method omitted*/
}

public class box extends Figure {
    private double _v1;
    private double _v2;

    public box(double length, double width) {
        _v1 = length;
        _v2 = width;
    }
    public double length() {
        return v1;
    }
    public double width() {
        return v2;
    }

    /*equals and toString method omitted*/
}

public class circle extends Figure {
    private double _v1;

    public circle(double radius) {
        _v1 = radius;
    }
    public double radius() {
        return _v1;
    }

    /*equals and toString method omitted*/
}
```

⋮

Short Record Type Definitions

Short record definitions are short hand for variant definitions with only one alternative. Thus, the translation of a short record definition is very similar to the one for a variant definition. The destructor and reconstructor of a short record definition are optional like for a variant definition. Since the short record definition has only one alternative, there is no need for an abstract class and a class inherited from this.

The translation from RSL into Java is as follows:
RSL:

type t :: get_c_1 : t_1 get_c_2 : t_2 $\leftrightarrow$ replace_c_2,

Java:

```
public class T {
  private T1 _v1;
  private T2 _v2:

  public T(T1 _v1, T2 _v2) {
    this._v1 = _v1;
    this._v2 = _v2;
  }

  public T1 get_c1() {
    return _v1;
  }

  public T2 get_c2() {
    return _v2;
  }

  public T replace_c2(T2 _c2) {
    return new T(_v1, _c2);
  }
}

//Translation of t1 and t2 omitted
```

T is the translation of the name t.
T_1 is the translation of the type t_1.
T_2 is the translation of the type t_2.

An example of translation of a short record definition is shown in Example 2.28.

Example 2.28 Translation of a short record definition.

```
scheme SHORT_RECORD =
class
  type
    MySRD :: sel_Real : Real sel_Int : Int
end
```

```
public class MySRD{
    private double _v1;
    private int _v2;

    public MySRD(double _v1, int _v2) {
        this._v1 = _v1;
        this._v2 = _v2;
    }

    public double sel_Real() {
        return _v1;
    }

    public int sel_Int() {
        return _v2;
    }

    /*equals and toString method omitted*/
}
```

Union Type Definition

Union definitions are short hand for variant definitions. They have been excluded from the translation, because they allow omitting constructors of the underlying variant type. This makes it difficult to determine which alternative to use at a given point, because it is not explicitly stated. The union definitions are only short hands for variant definitions, therefore the removal of union definitions does not remove any power of RSL. It does, however, make writing specifications a little more tedious.

Abbreviation Type Definition

An abbreviation definition consists of an identifier and a type expression describing the type. The identifier can be used as a reference to the type expression. The abbreviation type gives a kind of encapsulation of the details concerning the type. The idea of a class in Java is to be an encapsulation of details, this is the reason that a class in Java is chosen as translation for an abbreviation definition.

Example 2.29 Abbreviation definition

type
  newType = t1 × t2,
  t1,
  t2

```
public class newType{
  public t1 _v1;
  public t2 _v2;
 /*equals and toString method omitted*/
}

public class t1 {
/*equals and toString method omitted*/
}
public class t2 {
/*equals and toString method omitted*/
}
```

2.3.4 Subtypes

In RSL a type is a subtype of another type, if the former is a subset of the latter. The rules for type checking in RSL dictate that the type checking

should only consider the maximal types according to a set of rules for determining maximal types. Therefore, the specification in Example 2.30 does not violate the rules for type checking in RSL. The concept of applying a method with an argument which is of a wider type than allowed cannot be done in Java. This is the reason, that subtypes are not translated in this project.

Example 2.30 Specification applying a maximal type to a function of a subtype.

```
scheme SUBTYPE =
class
   value
      function_a : Nat → Bool
      function_a(a) ≡ true,
      function_b : Nat → Bool
      function_b(b) ≡ function_a(0−b)
end
```

However, one could imagine a solution for subtypes using the assertion feature in Java. The solution would involve inserting an assertion statement before and after each use of a statement or an expression involving a field of a subtype, a method taking a subtype as argument, or a method returning a subtype. The assertion statement should use a translation of the subtype predicate specified as assertion expression. In Example 2.31, an example of translation of the use of **Nat** is shown.

2.4 Values in RSL

The following section is concerned with the values in RSL, how to define values using value definitions, and how to express them using value expressions. First the value definitions are described and afterwards the value expressions are described along with their translation in Java and possible alternatives are considered.

2.4.1 Ordering of Variables and Values

There is a challenge, when translating explicit value definitions and variable declarations in RSL, which are translated as fields in a class in Java. Both

Example 2.31 Translation of a subtype using assertions.

variable
  val : **Nat** := 1
test_case
  [t1] val := 2,
  [t2] val

For an explanation of **test_case** see Section 2.8.

```
public static int val = 1;

public static void main(String[] args) {
    assert (val > 0);
    System.out.println("[t1]: ");
    val = 2;
    assert (val > 0);
    System.out.println("[t2]: " + val);
}
```

in RSL and in Java it is possible to define these values or fields in terms of each other, see Example 2.32.

Example 2.32 Ordering problem for values and variables.

variable
  i : **Real** := r
value
  r : **Real** = s + 1,
  s : **Real** = 5.4

```
public class Ordering_1 {
    public double i = r;
    public static final double r = s + 1;
    public static final double s = 5.4;
}
```

In Java, the order of fields in a class does not matter, except when they are initialized at the time of creation. The Java source code presented in Example 2.32 is not valid, because the initialization of the first field is depending on the second field, and the value of the second field is not known at the time of creation of the first. The variables may be initialized using the statements shown in Example 2.32 but they need to be ordered in another way to be valid. The correct ordering is shown in Example 2.33.

Example 2.33 Correct ordering for the field in Java.

```
public class Ordering_2 {
    public final double s = 5.4;
    public final double r = s + 1;
    public double i = r;
}
```

2.4.2 Value Definitions

The idea of a value definition is to define a value and give it a name by which it may be referenced elsewhere in a specification.

In RSL, it is possible to overload these values, i.e., to use a name more than once for different values. Java also has the possibility of overloading, but only methods may be overloaded. The overloaded versions of a method in Java must have different parameters. A different return type is not enough. It is therefore not possible to reuse all names in RSL directly in Java.

Possible solutions to this problem include:

1. Disallow overloading in specifications to be translated.

2. Allow overloading in RSL which corresponds to the overloading allowed in Java.

3. Allow overloading in RSL, and use a naming convention in the translation into Java.

The second suggestion has been chosen, because it allows for some kind of overloading, but it does not present a problem in the implementation. The third solution could also have been chosen, but it would require the translator to keep track of identifiers and their types, and then, based on the type of the context, insert the correctly renamed identifier in Java.

Explicit Function Definitions

An explicit function definition in RSL is a function which is specified using a value expression for determining the outcome of using the function. It is explicitly specified how the to get the result from the function parameters and values in the specification. For an explicit function definition to be translated, it must only use value expressions for which translation are defined.

An explicit function definition in an RSL specification corresponds to a method definition in Java. They both take values as parameters and may

return some kind of value.

An explicit function definition has the form:

single_typing formal_function_application ≡ value_expression [pre_condition]

The single typing must be a binding using an id or an op identifier and a type expression. The maximal type of the type expression must be a function type [11]. If the identifier is an id the formal function application of the explicit function definition must be an id application.

In this work, only functions using id applications are considered because prefix and infix applications are intended for overloading and redefining the built–in operators, which Java does not allow. A translation of prefix and infix applications could be to create the methods corresponding to the function and let expressions using the redefined prefix or infix expressions be translated as method invocations of methods created.

For an id application it must hold that there are the same number of identifiers in the formal function parameter list as there are type expressions in the first part of the function type expression. This simply means that there must be an identifier for each of the types in the argument of the signature of the explicit function definition.

These limitations on the functions in RSL ensures that functions have the form which is shown in the translation.
RSL:

id : type_expr$_1$ × ... × type_expr_n → type_expr$_{n+1}$
id(binding$_1$,...,binding$_n$) ≡ value_expr

Java:

public [**static**] T_{n+1} id (T_1 b_1 ,... ,T_n b_n) V

T_i is the translation of type_expr$_i$.
V is the translation of value_expr.
b_1 to b_n are the translation of binding$_1$ to binding$_n$. Whether or not the keyword **static** is added depends on the context, a discussion of this can be found in Section 2.2.6.

The translation of value expressions are dealt with in Section 2.4.3.

As shown in the Example 2.34, a number of variables are created in the translation, which we can clearly see are not necessary. The closest translation is shown in Example 2.35.

Example 2.34 Listsum translation.

listSum : **Int*** → **Int**
listSum(il) ≡
  if il = ⟨⟩ **then**
    0
  else
    hd il + listSum(**tl** il)
  end

```
public static int listsum(RSLList<Integer> il) {
    int _v0 = 0;
    if(il.equals(new RSLListDefault())) {
        int _v1 = 0;
        _v1 = 0;
        _v0 = _v1
            }
    else {
        int _v1 = 0;
        _v1 = il.hd() + listSum(il.tl());
        _v0 = _v1
            }
    return _v0;
}
```

Example 2.35 Listsum translation closest to specification

```
public static int listsum(RSLList<Integer> il) {
    if(il.equals(new RSLListDefault())) {
        return 0;
    }
    else {
        return il.hd() + listSum(il.tl());
    }
}
```

The reason for using the first translation is that for nested constructs using blocks, there is not always a return statement in each branch, which is required in Java. Therefore, a variable declaration statement is inserted in the top of a block in a method and this statement is returned at the end of the block. Constructs creating new blocks like the if–else statement inserts a new variable in the top of the blocks created and an assignment statement a the end of the blocks. The assignment statement assigns the local variable in the block to the variable of the outer block.

Implicit Function Definitions

Implicit function definitions are as the name suggest, implicit, and therefore not considered in this project. There may, however, be a number of special cases in which an implicit function definition can be translate, but these are also not considered here.

Explicit Value Definitions

In RSL, an explicit value definition defines and gives name to a constant value in RSL, i.e. a value, which cannot be changed. The value may be referenced in the definitions of other values or functions. An explicit value definition in RSL has the form:

single_typing = value_expr

The way to represent a value in Java, which may be referenced by methods and other values is as a field in a class. To make the value unchangeable the keyword **final** is added. This project does not consider product type expression in other places than as part of a function type expression.

Therefore the definition has the form shown in the translation.
RSL:

id : type_expr = value_expr

Java:

```
public [static] final T Id = v;
```

T is the translation of type_expr.
V is the translation of value_expr
Id is determined from the identifier id in RSL.
Whether or not the keyword **static** is added depends on the context, a discussion of this can be found in Section 2.2.6.

An example of the translation of explicit value definitions is shown in Example 2.36.

Example 2.36 Translation of explicit value definitions.

value
  constant_one : **Int** = 1,
  constant_two : **Text** = ″a″

```
public static final int constant_one = 1;
public static final String constant_two = "a";
```

Implicit Value Definitions

Implicit value definitions are like implicit function definitions, not considered in this project. Like for implicit function definitions, there may be a number of special cases where the implicit definition carries enough information to be translated.

2.4.3 Value Expressions

In RSL, there is only one way to describe a value and that is by a value expression. In Java there are three levels of constructs for describing values:

1. Blocks.
2. Statements.
3. Expressions.

The problem with different number of levels becomes evident when dealing with a value expression in RSL containing other value expressions. As an example, an if–then–else expression in RSL contains a value expression as condition. The condition determines whether to select the first or the second branch of the if–then–expression. The translation of an if–then–else expression in RSL is, as stated later, an if statement in Java. The if statement in Java must have an expression as condition. When a specification in RSL uses an if–then–else expression as condition of another if–then–else expression, then the outer if–then–else expression is translated as an if statement in Java. The condition cannot be translated as another if statement, because the condition of the outer if statement in Java must be an expression and not a statement. This is shown in Example 2.37.

Example 2.37 Not allowed translation of nested **if–then–else** expression.

if if a = b **then false else true end then**
c
else
d
end

```
if(if(a == b) return false; else return true;)
    return c;
else
    return d;
```

One possible solution of the problem shown in Example 2.37 could be to rewrite the condition using the conditional expression in Java `a == b ? false : true`. This does solve the problem in the general case, because other type of statements does not have an expression counterpart. A more general solution could be to translate the condition as an auxiliary method returning a boolean value, and let the translation of the condition of the outer if statement be a method invocation of the auxiliary method created. The idea is shown in Example 2.38.

Example 2.38 Allowed translation of nested **if–then–else** expression.

if if a = b **then false else true end then**
c
else
d
end

```
if(_condition1())
    return c;
else
    return d;

private boolean _condition1() {
    if(a == b)
        return false;
    else
        return true;
}
```

The general strategy for dealing with the fact that Java has three levels for expressing values, while RSL only has one, is to allow any value expression in RSL to be translated to either of the levels in Java.

A *block* in Java is a possibly empty collection of statements. There exists a special statement in Java, namely the expression statement, which turns an expression into a statement.

If a block is needed in the translation and the direct translation of the RSL value expression is a Java expression, then the Java expression can be wrapped in an expression statement, and the expression statement can be wrapped in a block. The problem is more complicated if a RSL value expression is translated into a Java statement, and the nesting construct in Java only allows for a Java expression. In order to solve this, the statement, which is the translation of the RSL value expression is, placed in an auxiliary method, and a method invocation of the auxiliary method is inserted instead. A method invocation is an expression in Java. One could imagine three methods, which all take a RSL value expression as parameter and return one of the Java constructs:

- `make_block(RSLValueExpression e)`
- `make_statement(RSLValueExpression e)`
- `make_expression(RSLValueExpression e)`

The function `make_block` either translates the *RSLValueExpression* or calls `make_statement` and wraps the result into a block. The function `make_statement` either translates the *RSLValueExpression* or calls `make_expression` and wraps the result in an expression statement. If `make_expression` is called with a *RSLValueExpression*, which is translated as a statement in Java, it inserts a method invocation and creates an auxiliary method containing the translation of the *RSLValueExpression*. If `make_expression` is called with an RSL expression which is translated as an expression it just returns an expression.

The following sections deals with the translation of different value expressions in RSL.

If–then–else Expressions

An if–then–else expression in RSL correspond very much to an if statement in Java. They both contain an expression, which must evaluate to a boolean value determining whether to select the first or second alternative. In RSL it is possible to list a more than two alternatives by using the **elsif** keyword and specifying the condition for the alternative. An if–then–else expression

using **elsif** construct can be rewritten using only **if then else**, see Example 2.39 [11]. An if statement in Java is chosen as translation of an if–then–else expression in RSL.

Example 2.39 Rewrite elsif construct.

```
if a = b then
   0
elsif a = c then
   1
   elsif a = d then
   2
else
   3
end
```

Is equivalent to

```
if a = b then 0 else
   if a = c then 1 else
      if a = d then 2 else
         3
      end
   end
end
```

```
if(a == b)
    return 0;
else if(a == c)
    return 1;
else if(a == d)
    return 2;
else
    return 3;
```

Case Expressions

Case expressions in RSL allow for a choice between several alternatives [11]. A case expression consists of a value expression and a list of case branches. A case branch consists of a pattern and a value expression. By evaluating the value expression of the case expression and matching it with the patterns in the case branch list, a branch may be selected, and the corresponding value

expression is evaluated. This description seems very similar to the switch statement Java, which also consists of an expression, that is compared with the value of the cases. If one of the values is matched the corresponding block of the case is evaluated. Here the similarities between the switch statement in Java and the case expression in RSL stop, because the switch statement in Java can only have patterns of simple types, i.e., **char**, **int** or **double**. The case expression of RSL can have literal patterns, name patterns, record patterns, list patterns, product patterns, and wildcard patterns allowing for a large variety of tests. Therefore, the translation of the case expression in RSL cannot be a switch statement in Java, but is instead a set of if statements in Java using different conditions depending on the pattern of the case expression.

Only name patterns, literal patterns, and record patterns of variant definitions are considered here. The translation of the patterns will be covered in the following. Example 2.40 shows the translation of a case expression using only name patterns. The names must be names of variant constants and due to the translation of variant constants as classes, the match is done using an **instanceof** expression in Java.

In Example 2.41 the translation of a case expression in RSL using both name patterns and a simple record pattern is shown. The name pattern is treated in the same way as shown in Example 2.40. The record pattern is treated a little different. The condition of the if–statement in the translation of a record pattern is the same as for the name pattern. In case of a match the first part of the if–block in Java is to define some local variables according to the names specified in the record pattern. The variables are set to the values of the variant by calling the destructors of the variant.

Example 2.42 shows the translation of a case expression in RSL using value literals and a wildcard pattern. The condition for the value literal pattern uses an infix expression with the Java equivalence operator == or a method invocation of the equals method. A wildcard pattern is translated as an else–block of the last if–statement.

Value Literals

Value literals in RSL are translated by creating a corresponding value literal in Java, according to the translation of the built–in types specified in Section 2.3. As an example a value literal of type real in RSL is translated into a value literal of type double in Java. This simple way of just creating a corresponding value literal could create problems with sizes of types according to the discussion in Section 2.3. One solution could be to add a kind of validation of a value literal in RSL before creating a corresponding value literal

Example 2.40 Translation of a case expression using name patterns.

```
scheme CASETEST_1 =
class
   type
      Colour == Black | White
   value
      invert: Colour → Colour
      invert(c) ≡
         case c of
            Black → White,
            White → Black
         end
end
```

```
/* Translation of Colour is omitted, but should
   be translated as stated for short record
   type definitions.
*/
public class CASETEST_1 {
    public static Colour invert(Colour c) {
        Colour _v0 = null;
        if(c instanceof Black){
            Colour _v1 = null;
            _v1 = new White();
            _v0 = _v1;
        }
        else { if(c instanceof White) {
            Colour _v1 = null;
            _v1 = new Black();
            _v0 = _v1;
        }
        }
        return _v0;
    }
}
```

Example 2.41 Translation of a case expression using record and name patterns.

```
scheme CASETEST_2 =
class
  type
    MyList == Empty | Add(head : Int, tail : MyList)
  value
    addOne: MyList → MyList
    addOne(ml) ≡
      case ml of
        Empty → Empty,
        Add(i, m1) → Add(i + 1, addOne(m1))
      end
end
```

```
/* Translation of MyList omitted in example,
   but is translated as stated for variant
   type definitions.
*/

public class CASETEST_2 {
    public static MyList addOne(MyList ml) {
        MyList _v0 = null;
        if(ml instanceof Empty) {
            MyList _v1 = null;
            _v1 = new Empty();
            _v0 = _v1;
        }
        else {if(ml instanceof Add) {
            MyList _v1 = null;
            int i = ((Add) ml).head();
            MyList m1 = ((Add) ml).tail();
            _v1 = new Add(i + 1, addOne(m1));
            _v0 = _v1;
        }
        }
        return _v0;
    }
}
```

Example 2.42 Translation of a case expression using value literal patterns and a wildcard pattern.

```
scheme CASETEST_3 =
class
  value
    toString: Int → Text
    toString(i) ≡
      case i of
        1 → "One",
        2 → "Two",
        _ → "Unknown"
      end
end
```

```
public class CASETEST_3 {
    public static String toString(int i){
        String _v0 = null;
        if(i == 1) {
            String _v1 = null;
            _v1 = "One";
            _v0 = _v1;
        }
        else {
            if(i == 2) {
                String _v1 = null;
                _v1 = "Two";
                _v0 = _v1;
             }
            else{
                String _v1 = null;
                _v1 = "Unknown";
                _v0 = _v1;
            }
        }
        return _v0;
    }
}
```

in Java, to ensure that the limitations of the type in Java are kept.

Application Expressions

Application expressions in RSL can result in a number of different translations depending on the context of the application expression. The application expression consists of two parts a value expression, which must be the name of a function, a list, a map, a make function of a short record definition, a constructor of a variant or a reconstructor, and a second part which is an optional expression list.

The creation of new records is translated using the constructor defined in the translations of type definitions. The use of destructors and reconstructors is translated as dynamic method invocations according to the translations of them. The application expressions involving list and maps have already been described in Section 2.3.2. Application of a function defined via explicit function definition in RSL is translated as a method invocation of a method in Java.

One could argue that using different translations is not a good solution, and that the constructs should have been translated, so that all application expressions could be treated identically. The drawback of that solution would be that the translated Java source code would not fulfill the requirements for good code practice in Java and not be very object–oriented. Finally the application expression concerning maps and lists would always be different from the application of the functions, so some kind of handling of different translations of application expressions would be needed.

Examples of these application expressions and their translation is shown in Example 2.43.

Bracketed Expressions

In RSL, a bracketed expression is meant for deciding the order in which an expression is evaluated. The same is the case for a parenthesized expression in Java. Parentheses in Java may be used to ensure that the content of the parentheses are evaluated before the context in which the expression appear. The translation of a bracketed expression in RSL is therefore a parenthesized expression in Java.

Basic Expressions

A basic expression in RSL consists of either of the keywords:

- **chaos**

Example 2.43 Translation of record constructors, make function, destructors and reconstructors.

```
type
   MyVariant == Alternative1(getReal : Real ↔ setReal),
   MyShortRecordDefinition :: getInt : Int ↔ Int
value
   listSum : Int* → Int
   listSum(il) ≡
      if il = ⟨⟩ then 0 else hd il + listSum(tl il) end
test_case
   [t1] getReal(Alternative1(5.0)),
   [t2] setReal(5.0, Alternative(4.0)),
   [t3] getInt(mk_MyShortRecordDefinition(1)),
   [t4] setInt(2, mk_MyShortRecordDefinition(1))
   [t5] listSum(⟨1,2⟩)
```

```
/* Translations of MyVariant, MyShortRecordDefinition and listSum
     are omitted*/
public static void main(String[] args) {
     System.out.println("[t1]: " +
          (new Alternative_1(5.0)).getReal());
     System.out.println("[t2]: " +
          (new Alternative_1(4.0)).setReal(5.0));
     System.out.println("[t3]: " +
          (new MyShortRecordDefinition(1)).getInt());
     System.out.println("[t4]: " +
          (new MyShortRecordDefinition(1)).setInt(2));
     System.out.println("[t5]: " + listsum(
        new RSLListDefault<Integer>(new Integer[]{1,2}))));
}
```

- **skip**
- **stop**
- **swap**

chaos in RSL represents non–determinism. The concept of non–determinism is not wanted in a program, no translation of this into Java has been defined.
skip is part of the imperative specification style, which is not considered in this project.
stop represents a deadlock in a system which is involved in communication. Communication between parallel systems are not considered in this project, and thus **stop** is not translated.
swap "is a completely under–specified expression — it may terminate or not, may deadlock, may behave non–deterministically" [11, p. 189]. There is no possible translation of **swap** in Java.

Since neither of the basic expressions are considered in this project the use of basic expression is disallowed.

Statement Infix Expressions

Statement infix expressions are not considered here because the operators involved are part of the specifications of parallel systems which are not considered in this project.

Axiom Infix Expressions

In RSL, an axiom infix expression is an expression using an infix connective. Use of the infix connective and their translation are covered in Section 2.3.1.

Value Infix Expression

The translation of a value infix expression in RSL can result in two different kinds of expressions in Java, depending on the operator in RSL. The simple mathematical infix operators considered here, + and *, are both translated as infix expressions in Java using the same simple operators according to the discussion in Section 2.3. The concatenation operator $\frown$ in RSL is translated as a method invocation of the concat method in Java. The left–hand side operand is object of the method invocation, and the right–hand side operand is argument to the method.

The = operator in RSL comparing two values is translated in two different ways depending on whether the type of translation of the operands is a primitive type or a reference type in Java. In cases where the operands of a

value infix expression is translated as primitive types, the translation is an infix expression using the equivalence operator in Java (==). For reference types, the translation is a method call using the equals method of Java. A number infix expression and their translation are shown in Example 2.44

Example 2.44 Translation value infix expressions.

1+2
5∗6+2
⟨1,2,3⟩ ⌢ ⟨4,5,6⟩
1 = 1
"a" = "b"

```
1 + 2
5 * 6 + 2
(new RSLListDefault<Integer>(new Integer[]{1,2,3})).concat(
  new RSLListDefault<Integer>(new Integer[]{4,5,6}))
1 == 1
"a".equals("b")
```

Axiom Prefix Expressions

Axiom prefix expressions concern the prefix connective ~, and is treated in Section 2.3.1 dealing with type **Bool**.

Universal Prefix Expressions

Universal prefix expressions concern the universal state quantification operator which is a part of the axiomatic specification style which is not considered in this work.

Value Prefix Expressions

Value prefix expressions in RSL are translated as method invocations of a method in Java. The exact translation of each of the operators is treated in the sections on the types of the value expressions involved.

Disambiguation Expressions

Disambiguation expressions are not considered here because they are used to clarify typings in RSL in case of overloading resulting in expressions where the

type cannot be determined. Only simple forms of overloading are considered in this project, and in these cases disambiguation expressions are not needed.

List Expressions

Translation of list expressions are treated in Section 2.3.2.

Set Expressions

Translation of set expressions are treated in Section 2.3.2.

Map Expressions

Translation of map expressions are treated in Section 2.3.2.

2.5 Variable Definitions

Variable definitions are not considered in this project. They are part of the imperative specification style, which is not considered in this project. Suggestion for at translation is given in Chapter 7.

2.6 Axiom Definitions

Axiom definitions are not considered in this project. They primarily deal with the implicit specification style, which is not considered in this project.

2.7 Channel Definitions

Channel definitions are not considered in this project. They belong to a part of RSL, dealing with specifications of parallel systems, which is not considered in this project. Suggestions for a possible translation is given in Chapter 7.

2.8 Test Cases Definitions

Test cases in RSL is an addition invented by the people at UNU/IIST[3]. The basic idea of the test cases is to avoid having to write test code in a programming language, to which RSL is translated by a translator. A test

[3]United Nation University / International Institute of Software Technology

case in RSL consists of an optional identifier and a value expression. When translating a specification containing test case declarations, a main method is created. The test cases are translated as a call to the System.out.println method with an infix expression consisting of the identifier concatenated with the translation of the value expression in a parenthesized expression, as shown in Example 2.45.

Example 2.45 Translation of a test case.

test_case
[t1] ⟨1,2,3⟩

```
public static void main(String[] args) {
    System.out.println("[t1]: " + (
      new RSLListDefault<Integer>(new Integer[]{1,2,3})));
}
```

Chapter 3

Development Strategy

This chapter considers a number of topics in the process of designing and developing a tool for translation from RSL into Java. The tool is referred to as the *translator*. The chapter starts with an overview of some design options, which are used in the discussion of the architectural design and development process of the translator. Afterwards, the architectural design of the translator is presented, followed by an overview of the development process. The last section of the chapter presents the different subsets of RSL used in this work.

3.1 Design Options

The following describes two design methods which will be used in the discussion of the design of the translator in Section 3.2. The section describes the use of a bootstrapping process for developing a translator from scratch, and how to optimize a compiler using bootstrapping. Furthermore, the section presents the visitor design pattern. The descriptions are at a general level, if you are already familiar with these two methods of bootstrapping and the visitor design pattern, this section may be skipped.

3.1.1 Bootstrapping

The concept of bootstrapping originally refers to a story about a man being able to fly by pulling himself up in his bootstraps [6]. The term has been used later on to describe the idea of creating a new tool using the tool itself. The way it is typically done is by creating a simple version of the new tool by hand or other simple tools, and then use the simple version of the tool to create a more advanced version of a tool.

In the development of a language processor, the idea is to develop a language processor for a language using the language itself. First a primitive version of the language processor is created by hand, then this version is used for creating a more advanced version of the language processor.

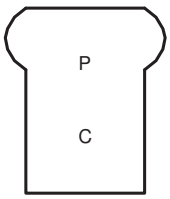
The term has been widened to the concept of processing a language processor using itself. The process may be used for creating a compiler from scratch, for optimizing a compiler, creating a true compiler from a portable compiler, or creating a compiler generating another target language. It is the development of a language processor from scratch and optimizing a language processor, which are of interest in this project and these two processes will be covered in the following.

Tombstone Diagrams

The application of a language processor on either programs or other language processors can be clearly illustrated by tombstone diagrams. The different kinds of diagrams will be briefly introduced in following. A more extensive covering can be found in [19].

An ordinary program is represented by a round-topped tombstone with the name of the program in the head of the tombstone and the implementation language program at the base of the tombstone, an example of a program is shown in Figure 3.1.

Figure 3.1 A program P implemented in C.



A machine, which may execute a program, is represented by a pentagon pointing downwards with the name of the machine code language it is able to execute inside of it, an example of a machine is shown in Figure 3.2.

An interpreter is represented by a rectangular tombstone with the language interpreted at the top of the tombstone and the implementation language at the base of the tombstone, an example of an interpreter is shown in Figure 3.3.

A translator or compiler is represented by a T–shaped tombstone with source language at the left, the target language at the right and the imple-

Figure 3.2 A machine running machine code M.

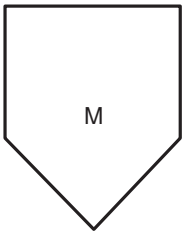
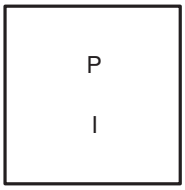
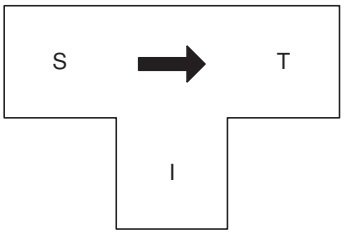


Figure 3.3 An interpreter interpreting P implemented in I.



mentation language at the base of the tombstone, an example of a translator is shown in Figure 3.4.

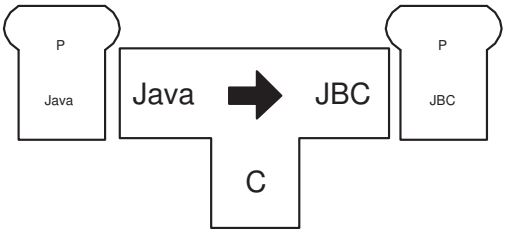
Figure 3.4 A translator translating from S into T implemented in I.



Translation of a program using a translator can be illustrated by putting the program to be translated to the left of the translator and the result to the right of the translator. An example of this is shown in Figure 3.5. The implementation language of the program to the left must match the source language of the translator. The implementation language is then changed by the translator to represent a program with a new implementation language to the right. If a machine or a combination of a machine and an interpreter can execute a language then a program with that implementation language may be executed.

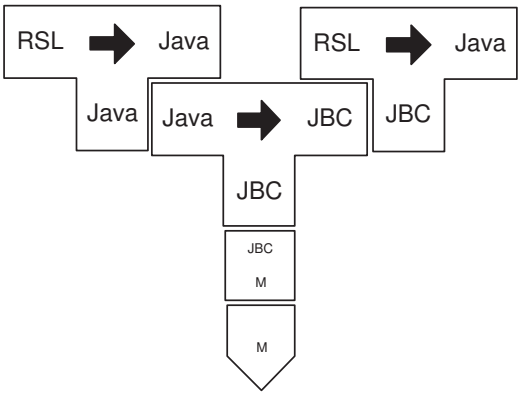
A translator is also a program and may be translated using a translator, which is a program and must be executed using a machine of a combination

Figure 3.5 A Java program P being translated to Java Byte Code (JBC) using a Java to JBC translator implemented in C.



of a machine and an interpreter. An example of this is shown in Figure 3.6.

Figure 3.6 An RSL to Java translator implemented in Java being translated into a RSL to Java translator implemented in JBC, using a Java to JBC translator implemented in JBC interpreted by a JBC to M interpreter on a machine M.

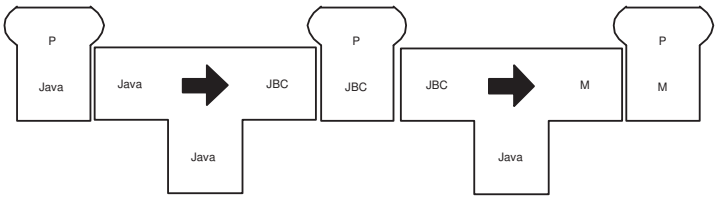


The tombstone diagrams can also be used to describe a multistage compiler by putting the compilers adjacent to each other separated by a program expressed in the intermediate language, an example of a multistage compiler is shown in Figure 3.7.

Full Bootstrapping Process

The full bootstrapping process is used for developing a new compiler from scratch. An advantage of the full bootstrapping process is that at the end of the process, the compiler is expressed in the source language, for which it is developed i.e. maintenance and extension of the compiler do not depend on any other languages. The development of the Java compiler will be used as

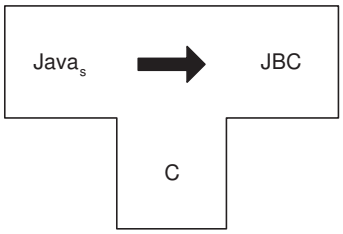
Figure 3.7 A two–stage compiler compiling from Java to M with JBC as intermediate language.



an example. The Java source code comes in different versions in this context. The first version is a small subset which is named Java$_s$. The target language of the compiler is JBC (Java Byte Code). An interpreter for JBC is assumed available, and therefore JBC is assumed to be executable.

The first step of the process is to develop a compiler for the small subset Java$_s$ in a language, for which a compiler is already available. The language used for developing the Java compiler was C [19]. The compiler is shown in Figure 3.8.

Figure 3.8 A compiler for Java$_0$ into JBC implemented in C.



This compiler for Java$_s$ is compiled using the available compiler for the implementation language. The result is an executable compiler for Java$_s$. The compilation is shown in Figure 3.9.

The second step of the bootstrapping process is to write a second version of the compiler. The second version is written in the source language Java$_s$, as shown in Figure 3.10. The second version of the compiler is then compiled using the first version, as shown in Figure 3.11.

The third and final step is to extend the second version of the compiler to the full language. The third version of the compiler is shown in Figure 3.12. The third version is compiled using the second version, as shown in Figure 3.13. The result is a compiler for the full source language written in some subset of the source language. There are no dependencies on other languages. The third version of the compiler may be used to compile new

Figure 3.9 Compilation of the developed compiler.

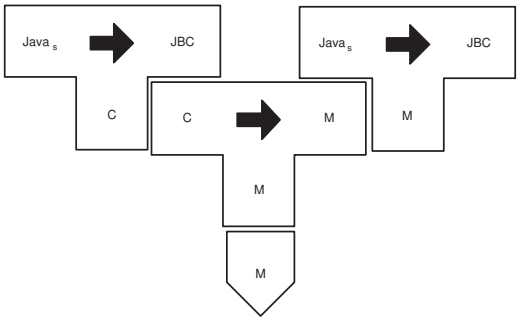


Figure 3.10 Second version of the compiler written in Java_s.

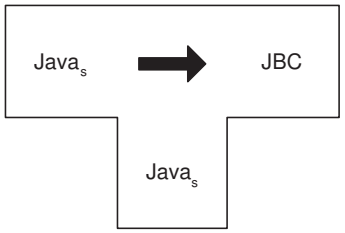
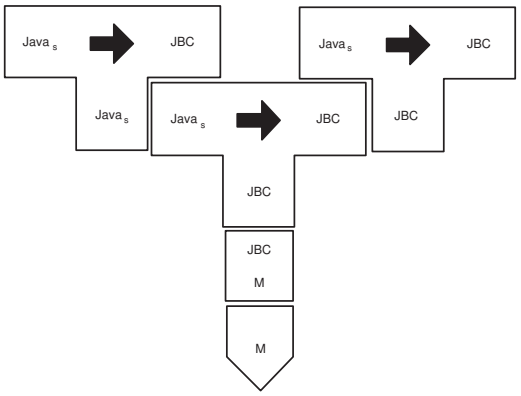


Figure 3.11 Compilation of the second version of the compiler.



versions of the compiler itself.

Optimization Using Bootstrapping

Bootstrapping can also be used to optimize a compiler. The optimization has two purposes:

Figure 3.12 Third version of the compiler extended to the whole source language.

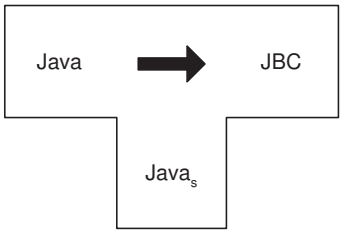
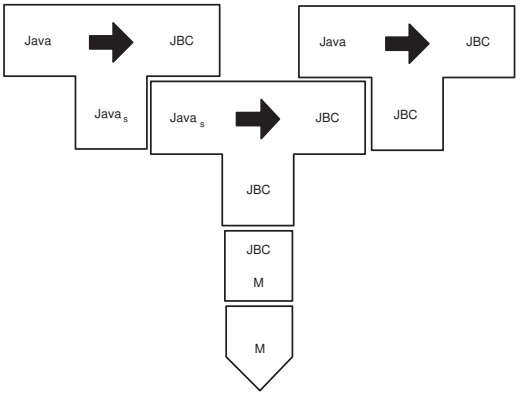


Figure 3.13 Compilation of the third version of the compiler.



1. Producing better programs, i.e. higher quality of code in the target language.

2. Improving the compiler itself, i.e. the efficiency in the compilation process.

The starting point for the optimization process is a compiler expressed in source language as well as in low quality target language. In this section optimization of an RSL to Java translator will be used as example. Java is assumed to be executable in this context. Both versions of the translators are shown in Figure 3.14. The quality of the target language is indicated by superscript *low* and *high* on the arrow of the translator and as subscript under the tombstone base.

The first step is to modify the first version of the translator implemented in the source language to produce higher quality target language, as shown in Figure 3.15.

The new version of the translator is translated using the executable version of the first translator, which is shown in Figure 3.16.

Figure 3.14 Left: Translator implemented in the source language. Right: Translator implemented in the target language.

Figure 3.15 The modified translator producing high quality target language.

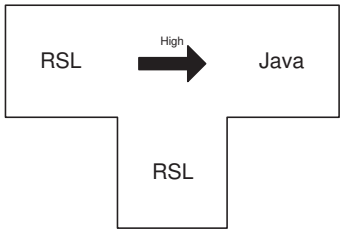
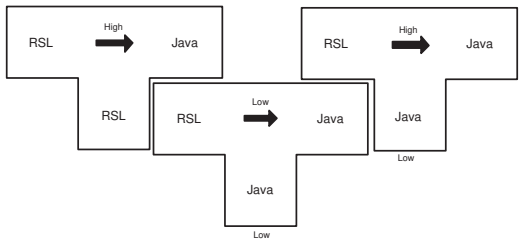


Figure 3.16 Translation of the new version of the translator using the first version.



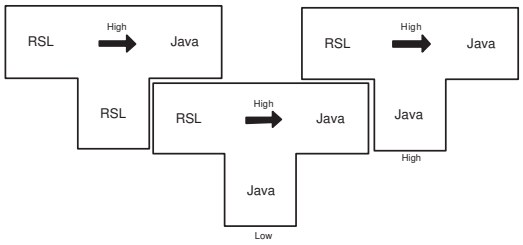
The result of the translation of the second version of the translator is a translator producing high quality target language. However, the executable version of the translator is still implemented in the low quality target language. This is fixed by translating the second version of the translator using the executable version of itself, as shown in Figure 3.17.

The result of this is a translator producing high quality target language implemented in high quality target language.

3.1.2 Design Patterns

"Each pattern describes a problem, which occurs over and over again, and then describes the core of the solution to that problem, in such a way that

Figure 3.17 Translation of the second version of the translation using the second version.



you can use the solution over and over again" [7, p. 2] In software design, this concept applies to several levels of abstraction.

At a low abstraction level, there is often a need for structures to represent data in some way. An example of this is the need to represent data as a list or as a tree. The structures are independent of the actual content of the structure. A list of integers is not very different from a list of real numbers. Therefore general solutions for representing different kinds of lists have been developed as well as methods for operating on these lists. Representation of different kinds of data structures and methods for operating on these structures have been described in numerous places [4].

At a higher level of abstraction, there may be a need for a solution for generating different output from the same input, or to ensure that only one instance of an object exists in an object–oriented language. Solutions for these kinds of design issues have also been described by several authors [7].

In the programming language chosen for this project, namely Java, there are libraries for many kinds of data structures as well as support for operating on them, i.e. solutions of the design issues of the first level of abstraction mentioned. No solutions are implemented for the higher level design issues. Therefore, solutions of these kind of issues must be implemented by hand. The following section deals with the visitor design pattern which will be used in the design of the translator.

Visitor Design Pattern

The purpose of the visitor design pattern is to separate a hierarchical structure of objects from tasks which must be performed on the structure [7]. The standard way to implement a feature in a structure is to add a method for the feature to each object in the structure. Each method carries out the feature for the object and invokes the same method in the subparts in the structure. As an example, a feature for printing a structure is implemented

by adding a print method to each object in the structure. Another feature of the structure is then implemented as another method in each of the objects. There are three problems with this solution:

1. Each feature is spread as methods in all the objects of a structure. This makes it difficult to get an overview of the feature.

2. Addition or removal of a feature requires changes in all objects of a structure. This makes changes in the functionality of a structure tedious.

3. The object hierarchy often represents a hierarchical structure in the real world. The structure has nothing to do with the tasks that must be performed on the hierarchy. It makes the design less clear that the tasks and the structure are mixed together.

The visitor design pattern solves this by moving the methods of the different features into separate task objects. The task objects are called *visitors*. A visitor traverses a structure of objects and completes the task by performing a part of the task while it visits an object. The visitor design pattern works by adding one method to all the classes in the object structure. The method is typically named accept or visit and takes a visitor as argument. The method calls a unique method, named visitObject, where Object is the name of the structure, in the visitor with itself as parameter. It is the unique method in the visitor which performes a part of the task.

One example of such an object structure could be a representation of a questionnaire in a system for doing interviews using a computer. A questionnaire consists of an introduction text and a number of questions. A question consists of a question text and a number of answer options. An answer option consists of a text describing the answer option. This can be represented by three types of objects: a questionnaire object, a question object, and an answer option object. A questionnaire must both be validated and printed, therefore each object contains a method for doing so. The questionnaire structure is shown in Figure 3.18.

In Figure 3.19, the same object hierarchy for an interviewing system is shown implemented using the visitor design pattern. The print and validate methods have been removed from the object structure. A class implementing the visitor interface has been added for each of the features.

Should the need arise for a new feature, it can be implemented as a third visitor in the second solution, whereas it would require addition of a method in each of the classes in the first solution.

The traversal of an object structure implementing the visitor design pattern can be placed in two places: either the traversal can be placed in the

Figure 3.18 Questionnaire–structure without the visitor design pattern.

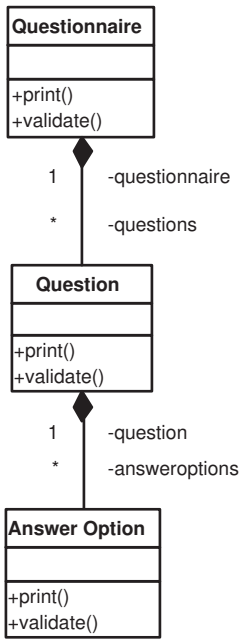
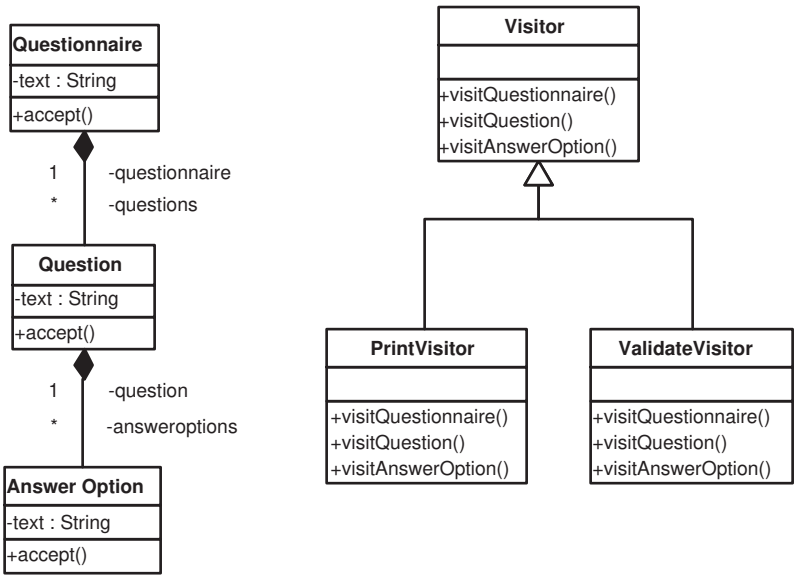


Figure 3.19 Questionnaire–structure with the visitor design pattern.



object structure in the accept method, or it can be placed in the visitors. The most simple solution is to keep the traversal in the object structure, because then it is kept in one place and not in all the different visitors. The reason for putting the traversal in the visitors anyway is in case that the traversal are different for the different visitors, or if it is necessary to perform actions, both before and after the traversal of the subcomponents in the visitObject methods.

A simple example of the need to perform actions both before and after the traversal of the subcomponents, is in case of a feature for printing a representation of a block in Java. If the traversal is put into the accept method of the block as shown in Example 3.1, then the call to visitBlock must be done either before or after the *for–loop* running through the statements. The problem with this is that if the call to visitBlock is done before the *for–loop* then there is no place to put the corresponding System.out.println("}"); statement and vice versa. The solution is to place the traversal in the visitor, i.e. the traversal of subcomponents is placed in the visitObject method for each object. This allows the visitor to perform actions both before and after the traversal of the subcomponents. In Example 3.2, the same example of printing a block in Java is shown with the traversal placed in the visitor.

Example 3.1 Visitor pattern – traversal in object structure.

```
public StringVisitor implements Visitor {
    public visitBlock(Block block) {
        System.out.print("{");
    }
}

public class Block implements Element {
    ArrayList<Statement> statementList;

    public void accept(Visitor visitor) {
        visitor.visitBlock(this);
        for(Statement statement : statementList)
            statement.accept(visitor);
    }
}
```

3.2 Architechtural Design of the Translator

The development of a translator between two languages is a complex task. Therefore, the task must be decomposed into smaller parts which can be

Example 3.2 Visitor pattern – traversal in visitor.

```
public StringVisitor implements Visitor {
    public visitBlock(Block block) {
        System.out.print("{");
        for(Statement statement : block.getStatementList())
            statement.accept(visitor);
        System.out.print("}");
    }
}

public Block implements Element {
    ArrayList<Statement> statementList;

    public void accept(Visitor visitor) {
        visitor.visitBlock(this);
    }
}
```

dealt with. The task of translating RSL into Java has been decomposed into three subtasks, each subtask is designed as a module in the translator.

1. Create an abstract representation of RSL based on the RSL given as input. (Front end)

2. Translate the abstract representation of RSL into an abstract representation of Java. (Translation module)

3. Generate Java source code based on the abstract representation of Java. (Back end)

The translation from an abstract representation of RSL into an abstract representation of Java is a considerable task. Based on that, it is decided to let the development of the translation module be incremental.

In order to make the development of the translation module incremental a full bootstrapping process is chosen. This makes it possible to maintain and extend the translation module using RSL, rather than a programming language or some other high level language. The use of a bootstrapping process in the development of one of the modules determines that this module should be implemented by hand in the languages involved in the bootstrapping process.

The next decisions are how to develop the front end and the back end of the translator. The front end may either be implemented by hand or developed using a tool, which generates a lexical scanner and a parser. The

latter solution is chosen, because it is easier to implement and maintain a lexical scanner and a parser in a high level language of a tool, rather than a lexical scanner and a parser implemented by hand.

There are no tools available for creating code based on an abstract tree representation. The abstract representation of Java implements the visitor design pattern. The generation of Java source code from an AST can therefore be implemented as a visitor.

The three modules involved in the translation process are controlled by a fourth module (Control module), which takes care of input and output as well as controlling the translation process.

To make the design clear, the interfaces between the different modules should be clear. There are four kinds of data, which flow in the translator, they are listed below:

1. RSL – One or more files written in the ASCII representation of RSL.

2. RSL AST (Abstract Syntax Tree of RSL)[1] – An instance of a class in Java along with the classes used by it.

3. Java Ast (Abstract Syntax Tree of Java) [2] – An instance of a class in Java along with the classes used by it.

4. Java – One or more files written in Java source code.

The interface between the modules are made clear by letting each of the modules take care of a transformation between two of the data structures.

The flow of translation is illustrated in Figure 3.20. The figure shows the three modules involved in the translation as well as the representations of data in the different stages of translation.

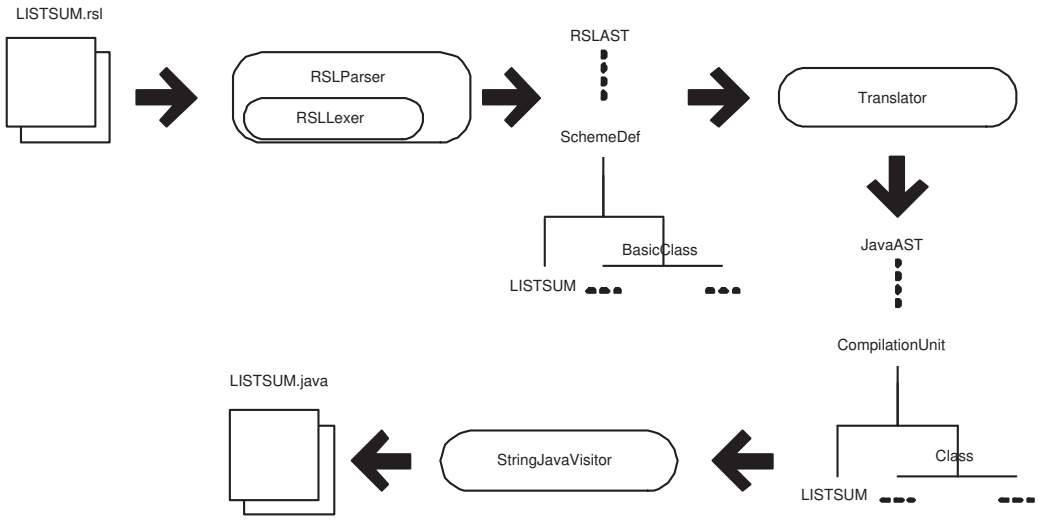
3.2.1 Data Structures

Data Structure RSL

RSL in this context is one or more files written in the ASCII representation of RSL according to [11]. Furthermore, the RSL used in the specification must be within the subsets described. The subsets are presented in Section 3.4.

[1]The term Abstract Syntax Tree (AST) is explained in Section 3.2.2
[2]The term Abstract Syntax Tree (AST) is explained in Section 3.2.2

Figure 3.20 Flow of translation.



Data Structure RSL AST

The data structure for representing RSL in the translator is an abstract syntax tree. The tree is implemented as a number of classes in Java, which all inherit from a base class containing a few common properties, namely a field for the parent node and a field for the type of the node. The data structure implements the visitor design pattern for traversing the tree. A concrete representation of an RSL specification is an instance of the top class. The structure of the RSL AST is based on the syntax found in [11, Appendix A]. Furthermore the definition of test cases has been allowed using the syntax described in [1].

Data Structure Java AST

The data structure for representing Java in the translator is also an abstract syntax tree. The tree is implemented as a number of classes in Java, which all inherits from a base class. This data structure also implements the visitor design pattern for traversing the tree. A concrete representation of Java is an instance of the top class of the hierarchy. The structure of the Java AST is based on the syntax found in [13, Appendix L] and [5].

Data Structure Java

Java in this context is one or more files written in Java source code. The source code generated uses some of the new features added in Java 1.5.0. In

Figure 3.21 Lexical analysis of if–then–else statement in RSL.

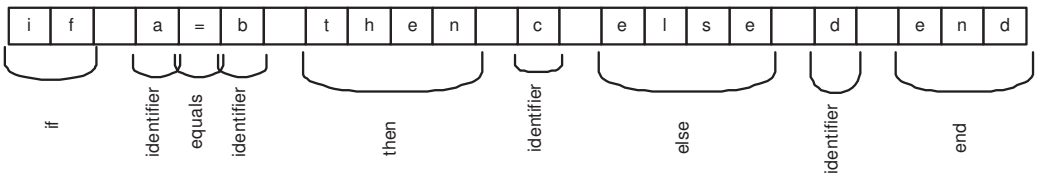
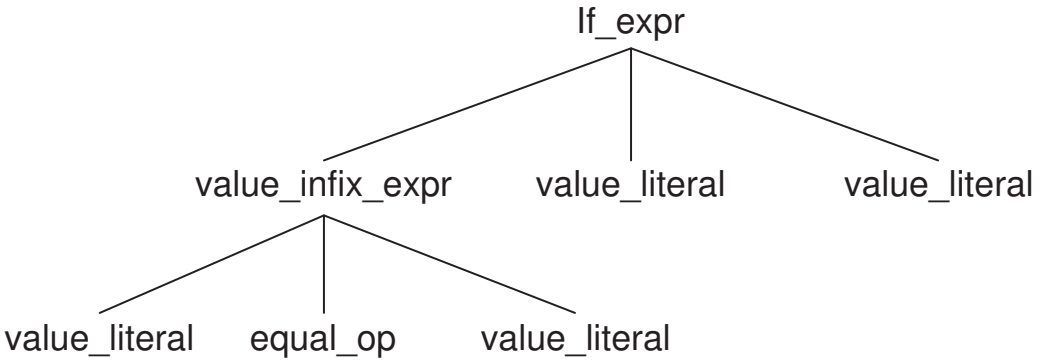


Figure 3.22 Parsing of if–then–else statement in RSL.



order to be able to use the output of the translator, one must have at least J2SE 5.0 (Java 1.5.0 SDK) or a later version installed.

3.2.2 Front End of the translator

As stated in Section 3.2, the front end should be developed using a tool. The front end in this context covers two phases of the translation process, namely lexical analysis of RSL and parsing RSL into an abstract syntax tree representation of RSL.

The lexical analysis is done by a lexical analyzer. A lexical analyzer, often referred to as a lexer, reads the input one character at a time and combines them into groups of characters having a collective meaning, known as tokens, an illustration of this is shown in Figure 3.21.

A parser creates an abstract syntax tree from the tokens passed on by the lexical analyzer. An abstract syntax tree is a tree representation of constructs in the source language where unnecessary information has been removed, this is shown in Figure 3.22. The keywords “if”, “then”, “else” and “end” has been removed. They are not necessary, because the node of the tree is if_expr node and it is the content of the if–then–else expression that is interesting.

Several tools were considered for the development of the front end. They

are presented in following section along with reasons for using or not using the tool.

Tools Considered

The tools considered are:

- Gentle [17]
- Lex & Yacc [8]
- SableCC [9]
- Javacc [3]
- ANTLR [14]

The first tool, Gentle, was considered, because several other tools for RSL have been developed using Gentle. These tools where developed entirely using Gentle. In this project, only a part of the translator should be developed using a tool. Gentle is not well suited for generating a part of a compiler, because Gentle generates a set of specifications, which are then processed by Lex & Yacc. The two phases of Gentle make it hard to see the correspondence between the specification provided for Gentle and the resulting source code generated by Lex & Yacc. The source code generated by Lex & Yacc is C. Therefore, some port of Gentle to Java was needed, in order for it to be part of a bootstrapping process with Java as target language. This should be possible since Lex & Yacc are ported to Java, but it represents a considerable task in itself.

The second option considered was Lex & Yacc, because they are almost synonymous with generation of compilers. The original tools were not considered, but ports of them to Java were. They were not selected, because of the better possibilities for specifying rules using EBNF[3] for the parser in some of the other tools.

SableCC was considered because it provides tools for traversing the AST generated. SableCC was not selected, because it is not possible to control the generation of the AST. Full control over the generation of the AST is needed to integrate between the front end and the bootstrapped translation module. The reason for this is that in order for a translation module specified in RSL to be type checked using the existing tools, the types used in the specification must be in the context of the specification, i.e. the data structures RSL AST

[3]See [6] p. 37.

and Java AST must be specified in RSL. Therefore, full control over the types used in the AST in the front end is needed.

The last two tools considered were Javacc and ANTLR. They are very similar in their interface. It is possible in both tools to specify rules using EBNF both for the lexical scanner and the parser. They both have a possibility for generating an AST and for adding action code to each rule in the grammar. ANTLR was preferred, because it is based on open source and it is more widely used than Javacc. ANTLR is furthermore better documented than Javacc.

ANTLR was chosen as tool for this work because it provides the best features.

3.2.3 Translation Module

The task of the translation module in the translator is to do a translation from an RSL AST into a Java AST. This is done in two steps.

1. A number of passes over the RSL AST to decorate it with types.

2. Create a Java AST based on a RSL AST.

Type Decorating the RSL AST

Before creating a Java AST the RSL AST has to be decorated with types. There are two reasons for this:

1. According to the translations specified in Chapter 2 several constructs in RSL must be translated to a number of different expressions in Java depending on the context and the parts involved.

2. The empty list requires a typing in Java.

An example of a construct, which requires different translations, is the application expression in RSL. In Example 3.3 a number of application expressions are listed in the test cases. The translation of these are presented in the listing of code in the example. The Java source code shows that the application of `mk_MyShortRecordDefinition` is translated as a class instance creation expression in Java, while the application of the function `myFunction` is translated as a method invocation of a method in Java. The application of `myList` is translated as a method invocation of a method on an object in Java. In order to handle these expressions differently, even though they are all application expressions, they must be decorated in some way.

Example 3.3 Different kinds of application expressions in RSL.

type
  MyVariantDefinition == Alternative1,
  MyShortRecordDefinition :: variant : MyVariantDefinition
value
  myList : **Text*** = ⟨″a″, ″b″⟩,
  myFunction : **Real** → **Real**
  MyFucntion(r) ≡ r + 1.0
test_case
  [1] mk_MyShortRecordDefinition(Alternative1),
  [2] myList(1),
  [3] myFunction(myFunction(5.0))

```
/* Translation of types and values omitted. */
public static void main(String[] args) {
    System.out.println("[t1]: " + new
      MyShortRecordDefinition(new Alternative1()));
    System.out.println("[t2]: " + myList.get(1));
    System.out.println("[t3]: " +
      myFunction(myFunction(5.0d)));
}
```

The passes over the RSL AST are done by a number of visitors. The first visitor passing over the RSL AST is a *ParentVisitor*, which takes care of setting the parent of each node in the RSL AST. One could argue, that this could have been done in the creation of the RSL AST in the front end. This would have lead to more complex action code in the grammar file of the front end. The second and third pass over the RSL AST is done by a *TypeDecorateVisitor*, which tries to determine the type of an expression, and, if successful, sets the type of the expression.

The reason that a typing of the empty list is required is somewhat complex. The translation of a list in RSL is an instance of the RSLListDefault<E> class in Java. The class in Java is a generic class which needs a typing. It is possible to use an untyped list if it is used as argument to a method in Java, but it is not possible to use an untyped list as argument to a constructor. Therefore all lists must be typed, even if they are empty.

In Example 3.4 an explicit function definition with a value expression containing two empty lists is shown. There is an empty list in the condition of the if–then–else expression, and there is an empty list in the if–branch of the if–then–else expression.

To determine the type of the empty list in the condition of the if–then–else expression, the *TypeDecorateVisitor* looks at the parent expression. The parent expression is a value infix expression using the equal operator. The syntax rules of RSL dictate, that both operands of a value infix expression must be of the same type if the operator is the equal operator. The type of the empty list in the condition can therefore be set to the type of the left–hand side operand. The type of the left–hand side operand is known, because it is a parameter of the function, where the type can be determined from the function type expression and the binding list.

To determine the type of the empty list in the if–branch the fact that each branch of an if–then–else expression must have the same type is used. The type of the else–branch of the if expression can be determined by determining the type of the value infix expression. The type of the value infix expression is determined by looking at the operator and the operands. In case of an operator for which the type of the result is the same as the operands, the type of the value infix expression can be set by determining the type of the operands. In Example 3.4 the operator of the value infix expression is the concatenation operator. The result of the concatenation operator is a list or a text, which is a list of **Char**. The operands of the concatenation operator must also be lists of the same type. The type of both value infix expression as well as the operands can be set, because of the type of the operands of the value infix expression is known. This means that the type of the else–branch and therefore the if–branch can be set.

Example 3.4 LISTMULT example.

listmult : **Int*** × **Int** → **Int***
listmult(il, i) ≡
  if il = ⟨⟩ **then**
    ⟨⟩
  else
    ⟨ **hd** il ∗ i ⟩ ⌢ listmult(**tl** il, i)
  end

```
public static RSLList<Integer> listmult(RSLList<Integer> il,
                                        int i) {
    if(il.equals(new RSLListDefault<Integer>())) {
        return new RSLListDefault<Integer>();
    }
    else {
        return ((new RSLListDefault<Integer>(il.hd() * i)).
                concat(listmult(il.tl(), i)));
    }
}
```

Creating a Java AST From a Decorated RSL AST

The translation from an RSL AST to a Java AST is done by a number of functions working from the top of the RSL AST to the bottom. The functions rely on recursion for running through the lists in the RSL AST. The reason for using recursion rather than iteration is that it is possible to use recursion both in Java and in the subset of RSL translated.

The top level function for doing the translation takes an RSL AST as argument and returns a Java AST. The functions works by calling other functions based on, what is needed at the place in the Java construct. A Java AST consist of a collection of compilation units. Therefore, two methods creating a number of compilation units from an RSL AST is called. The results of these two are joined and the Java AST is created. One of the compilation units is the translation of the scheme being translated. A compilation unit consists of a class declaration, which again for one part consists of a list of methods. The list of methods in the compilation unit is created by a method which runs through the declarations of the basic class expression of the scheme being translated and creates method declarations corresponding to the functions in the declarations.

The principle is that the structure of the components in the Java AST is known, and the functions create these structures based on the information

which may be found in the RSL AST. An example of this could be the translation of an explicit function definition. An explicit function definition is translated as a method declaration according to Section 2.4.2. A method declaration in Java consists of the following components:

1. A list of modifiers.
2. A name.
3. A return type.
4. A list of arguments
5. A block determining the output.

These fives items are needed to create a valid method declaration in Java. The top–level method for translating an explicit function definition does this by creating a method declaration and letting a number of auxiliary methods create the parts needed. One method creates a name for the method from the id of the explicit function definition in RSL. A second method creates the list of arguments for the method based on the binding list and the type expression in RSL. A third method determines the return type for the method from the type expression of the explicit function definition. A fourth method creates a block from the value expression of the explicit function definition.

3.2.4 Back End of the translator

The back end of the translator, i.e. the part transforming the Java AST to concrete Java source code is implemented by hand as a visitor. The Java AST implements the visitor design pattern for traversing the tree. Normally the back end of a translator performs code generation as well as optimization of the code, before it is written to files. In this solution the back end is only responsible for writing the Java source code from the Java AST. The optimization should be done in the translation module. The reason for not letting the back end do any changes to the code is that by keeping this in the translation module, the optimizations may be specified using RSL rather than implemented by hand in Java.

3.2.5 Control Module

The purpose of the control module is to bind the other modules together and to be interface to the users of the translator. The control module should instantiate the other modules and pass the data structures to the modules.

In order to make the implementation of the control module easy, it should complete one stage of the translation before starting the next stage.

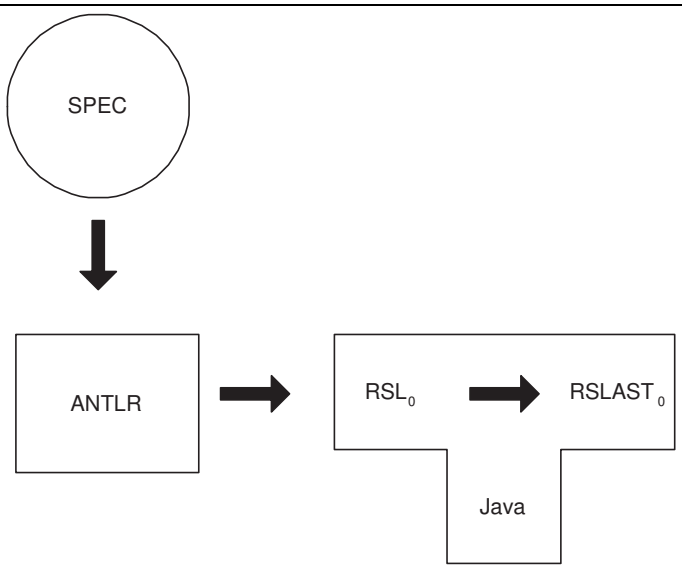
3.3 Development Process

The translation module of the translator is developed using a full bootstrapping process. This means that the translator will be developed stepwise. This section gives an overview of the steps and components involved in the bootstrapping process in the development of the translator. The translator is divided into three modules. Each module takes care of one stage in the translation process. The steps and modules will be illustrated using tombstone diagrams.

The development of the translator from RSL into Java source code assumes that there is a compiler from Java to Java byte code and an interpreter for the machine used available, i.e. it assumes that J2SE SDK is available for the platform. The translation from Java into Java byte code and interpretation of the byte code on the machine has been omitted from the diagrams. When a program or a translator has Java as implementation language it is assumed to be executable in this context.

The first step is to develop a translator which translates a small subset of RSL into Java. Figure 3.23 illustrates the use of ANTLR to create a trans-

Figure 3.23 Using ANTLR.



lator translating from RSL_0 to $RSLAST_0$, which is the AST representation of RSL_0.

Figure 3.24 Front end: Tool translating from RSL_0 to $RSLAST_0$.

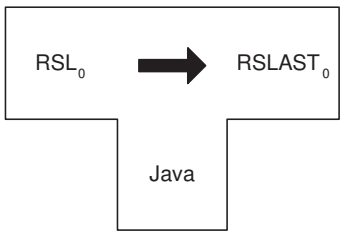


Figure 3.24 illustrates the result of using ANTLR on a specification of the syntax of RSL_0, i.e. the front end of the translator.

Figure 3.25 Translation Module: Tool translating from $RSLAST_0$ to JavaAst.

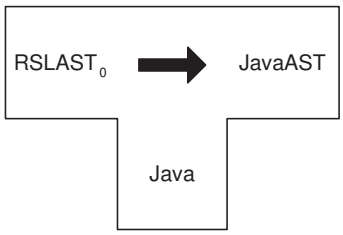


Figure 3.25 illustrates the translation module implemented by hand, which translates between the two abstract representations of the languages.

Figure 3.26 Back end: Tool for generating Java from the JavaAst.

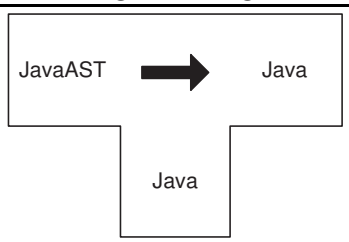


Figure 3.26 illustrates the module implemented by hand, which translates from the Java AST into Java source code, i.e., the back end of the translator.

These three modules can be combined into a three–stage translator. The flow of a program being translated is shown in Figure 3.27.

Figure 3.27 Combining the parts to a translator from RSL_0 to Java.

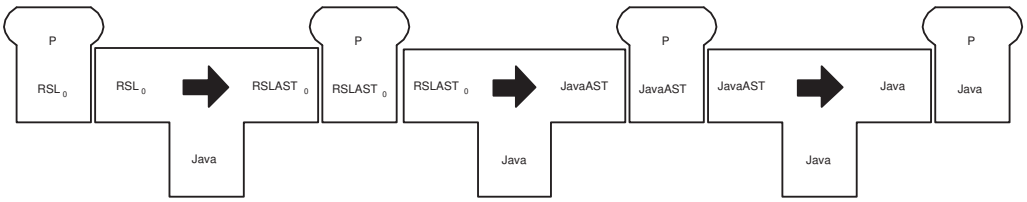
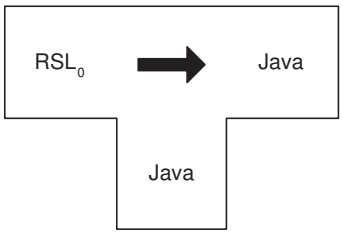


Figure 3.28 Result of combining the parts.



The three–stage translator can be viewed as one translator from RSL_0 to Java source code, as shown in Figure 3.28. This concludes the first step of the bootstrapping process. The result of this step is, as shown, a translator translating from RSL_0 into Java where each module is implemented by hand. This version of the translator is referred to as the "first version of the translator".

The second step of the bootstrapping process is to create a specification of a translation module from $RSLAST_1$ into JavaAST using RSL_0 as implementation language. The new module is shown in Figure 3.29, and the specification in RSL may be found in appendix C.1.1. Furthermore, the RSL AST must be modified to integrate with the new version of the translation module. At this point the $RSLAST_1$ could be extended to the full RSL language to avoid having to change it further in the next step. This was not done in this project, but it is suggested as the first point of an extension to this project in Chapter 7.

The translation module specified is made executable by first translating the specification developed using the translator developed in the first step. The translation is shown in Figure 3.30. This results in a new translation module translating from $RSLAST_1$ to JavaAST.

The new translation module is then combined with the modified front end and the back end into a new three–stage translator translating from RSL_1 into Java source code, as shown in Figure 3.31.

The new three–stage translator can be viewed as one translating from

Figure 3.29 Specification of a new translation module.

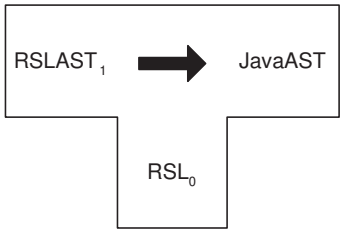


Figure 3.30 Translating the specification using the first version of the translator.

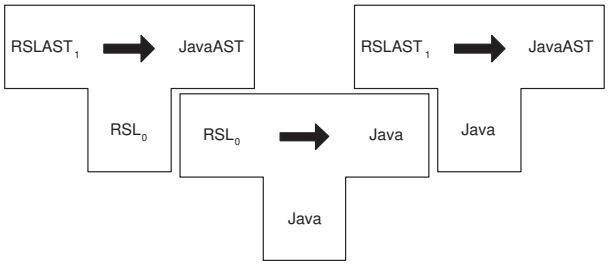
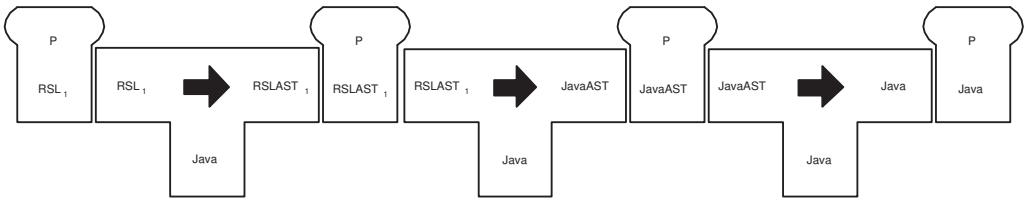
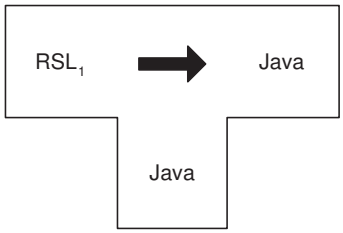


Figure 3.31 Combining the new translation module with the front end and back end.



RSL_1 to Java implemented in Java, see Figure 3.32.

Figure 3.32 Result of combining the new parts.



This concludes the second step of the bootstrapping process. The result is

a new translator which handles a slightly larger subset of RSL. This version of the translator is referred to as "the second version of the translator". The front end and the back end is only modified slightly by hand to fit the new constructs. The conclusion is that only a few modifications in Java are needed, while the main work in this step is done in RSL.

At this point, one can keep expanding and optimizing the translation module by specifying the new version in RSL_{n-1}. The specification can then be translated using the three–stage translator of the previous step. The translated specification can then be combined with the front end and the back end into a new three–stage translator translating from RSL_n to Java source code.

3.4 Subsets of RSL

The decision of using a bootstrapping process adds an extra dimension to the process of determining which subset of RSL to translate. It must be determined what the subset should be in the different steps of the bootstrapping process. To avoid doing too much of the implementation of the translation by hand the first subset of RSL, named RSL_0, should be as small as possible, yet strong enough to get the process started. It should be possible to specify a translation module using RSL_0.

3.4.1 Description of RSL_0

The RSL_0 subset of RSL only contains what is needed for some examples and for a specification of a translation module. The fact that the second version of the translation module must be specified in this subset sets some requirements of what is needed in RSL_0.

To be able to specify the translation module the types used in the translation module must also be specified in RSL. The types used in the translation module are the two AST representations of the languages. In order to specify the abstract syntax tree for a language, it is necessary to be able to define a type which has several alternatives as an example to specify that there is an infix operator which may be a number of different operators. Furthermore, one must be able to define that a construct consist of several other constructs, as an example to specify that an explicit function definition consists of among other things a type expression and a value expression. These two requirements mean that variant definitions are required in RSL_0. The variant definitions allow for a specification of several alternatives and the record constructors in variant definitions allow for a specification of that an

alternative may consist of several other constructs. To make variant definitions with one alternative simpler to define, short record definitions should also be included. Some constructs in RSL may contain a number of other constructs of the same type. As an example the variant definition may have a number of alternatives. To specify this lists must also be added to RSL_0. To handle simple examples the built–in types in RSL must be added, except for **Nat** which is a subtype of **Int**.

In order to be able to manipulate the two AST's defined, and to create one from the other, functions are needed. Therefore, explicit function definitions are part of RSL_0. Functions in a translation module are quite complex, so they must be able to take more than one parameter. Therefore, product type expressions in the first part of a function type expression is allowed. However, to make the implementation simple, multiple outputs from functions are not permitted. The outcome of an explicit function is described by a value expression. Therefore, a number of value expressions must be selected to define the outcome of functions.

In order to be able to make choices and differentiate between the different alternatives of a variant definition, case expressions and if expressions are part of RSL_0. Furthermore, it must be possible to manipulate the lists allowed in RSL_0, therefore prefix expression using the list operator **hd** and **tl** are also part of RSL. To cope with simple examples using the built–in types and to be able to combine lists and compare different value expressions, value infix expressions using +, *, ^ and = are part of RSL_0. In order to be able to call functions recursively, call other functions, construct records using constructors and make functions of short record definitions, application expressions are also part of RSL_0.

All these functions, and types which are needed in order to define a specification of a translation module must be put in some kind of module in order for them to be valid RSL. Furthermore, it would definitely be an advantage if the specification could be divided into several modules, because a specification of a translation module is quite large. These requirements mean that schemes and some kind of extend should be allowed in RSL_0, in order to divide the specification into several modules.

Overview of RSL_0:

- Top level constructs:
  - Schemes using the class expressions allowed. Parameterization not allowed.

  - Class expression:
    - Basic class expression using the declarations allowed.
    - Extending class expression with the base class as a scheme instantiation.
    - Scheme instantiations in an extending class expression.

- Type definitions:
  - Short Record definitions; destructors are required, reconstructors are not allowed.
  - Variant definition; destructors are required, reconstructors are not allowed.

- Value definitions:
  - Explicit function definitions using an id application and not products as result, using the value expressions allowed.

- Test case definitions using the value expression allowed. The test cases in RSL_0 must include an identifier.

- Type Expressions:
  - Function type expression. Only in form of total functions using the other type expression described as domain, and a function result description containing any of the other type expressions except product and function type expressions.
  - List type expression.
  - Product type expression as part of the domain of a function type expression.
  - Type literals excluding **Nat**.

- Value Expressions:
  - Value infix expressions using $\hat{}$, $=$, $+$ and $*$.
  - Value prefix expressions using **hd** and **tl**.
  - Enumerated list expressions.
  - Value or variable names.
  - Application expressions.
  - Structured expressions:

    - Case expressions using name, literal and record patterns
    - If expressions not containing **elsif** statements
  - Value literals.

3.4.2 Description of RSL_1

The RSL_1 subset of RSL is the next subset after RSL_0. This would be the point, where one tried to add as many features of the source language as possible. In this work, however, only one feature has been added namely the division operator for value infix expression. The reason for this is that the second step of the bootstrapping process in this project is only to prove that it is possible to get the process started. In Chapter 7 suggestions for how to extend the translator to a larger subset are given.

Chapter 4

Implementation of the Translator

This chapter describes some of the details of the implementation of the translator. It starts, in Section 4.1, with an introduction to the library classes developed for representations of lists, sets and maps. Afterwards, in Section 4.2 and 4.3, there is a presentation of the two versions of the translator implemented in this project.

4.1 The Library Classes

A number of library classes are implemented, which are common to all versions of the translator. These classes are used in the translation of three of the complex types in RSL, namely: Lists, sets and maps.

The classes and interfaces are grouped in a package named *translator.rsllib*, which is imported into the different versions of the translator. The package contains three interfaces and three classes:

1. *RSLList* – Interface, source code can be found in Appendix E.2.1.
2. *RSLListDefault* – Class, source code can be found in Appendix E.2.2.
3. *RSLSet* – Interface, source code can be found in Appendix E.2.3.
4. *RSLSetDefault* – Class, source code can be found in Appendix E.2.4.
5. *RSLMap* – Interface, source code can be found in Appendix E.2.5.
6. *RSLMapDefault* – Class, source code can be found in Appendix E.2.6.

Each of the three interfaces contains methods corresponding to the operators of the complex type it represents. The three classes are implementations of these interfaces.

The interfaces and classes relies on the generic feature introduced in Java 1.5, which makes it possible to define which type of objects that may be stored in a collection. This feature removes the need for type casting when retrieving an object from a collection. Another feature, which was added in Java 1.5 is boxing and unboxing of primitive types, which removes the need for wrapping a primitive type in a wrapper class, when an object is needed and unwrap a primitive type from a wrapper class, when a primitive type is needed.

Two listings of code are shown in Example 4.1. The two examples show an *ArrayList* of *Integer*'s. The first listing is written in code complying to Java 1.4, whereas the second listing is written in code complying to Java 1.5.

Example 4.1 Generic types, boxing and unboxing in Java 1.5.

Java 1.4

```
ArrayList al = new ArrayList();
al.add(new Integer(5));
al.add(new Integer(6));
System.out.println("al(0) + al(1): " +
  (((Integer)al.get(0)).intValue() +
  ((Integer) al.get(1)).intValue())
);
```

Java 1.5

```
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(5);
al.add(6);
System.out.println("al(0) + al(1): " + (al.get(0) + al.get(1)));
```

The three default classes are implemented using a different kind of collection in the *java.util* package for each of them. *RSLListDefault* is based on an *ArrayList*, *RSLSetDefault* is based on a *HashSet* and *RSLMapDefault* is based on a *HashMap*. All the classes are implemented so that a method returning a new complex type returns a modified shallow copy of the old one. A shallow copy in this context is a new instance of a class in Java containing references to the same objects.

The use of *ArrayList* as base for *RSLListDefault* is not an optimal solution, because the values in a list are not changed in RSL, therefore it is not necessary to create copies of a list – not even shallows copies to create the tail of a list. A linked list would be a more efficient choice. However, the

linked list must be implemented from scratch because the *LinkedList* class in the *java.util* package does not provide options for taking subsequences or the tail of a list. The *LinkedList* class in Java does implement a removeFirst method, but the method is destructive and cannot be used for implementing the **tl** operator on lists. A better solution would be an implementation of a linked list, in which it is possible to create a subsequence just by returning a reference to an object somewhere in the list. In Java, exists a class, *Vector*, which contains a method for creating a sublist, but the *Vector* class is based on an array, therefore, this not an effective solution either, for the problem described.

The implementation of the methods in the interfaces are straightforward and uses the methods defined in the underlying collections for most of the work. An example of the way of implementing the methods of the *RSLList-Default* class using the methods in the underlying collection is the implementation of the hd. hd uses the get method of *ArrayList* for returning the first element of the list. This method is listed in Example 4.2.

An example of the use of shallow copying is the tl method in the *RSLList-Default* class which creates a new *ArrayList* (list2) of the elements i.e. a new list of references to the elements in the old list. Then tl creates a new instance of *RSLListDefault* (result) and set the internal list of result to list2 and removes the first item of the list2 . The method is listed in Example 4.2.

Example 4.2 `hd` and `tl` methods in class: *RSLListDefault*

```
private ArrayList<E> list;

public E hd() {
    return list.get(0);
}

public RSLListDefault<E> tl() {
    ArrayList<E> list2 = new ArrayList<E>(list);
    RSLListDefault<E> result = new RSLListDefault<E>();
    result.list = list2;
    result.list.remove(0);
    return result;
}
```

One of the more complex methods in the library classes is the method compose in the *RSLMapDefault* class, which corresponds to the map composition operator in RSL. This method is listed in Example 4.3. The method does not use the generic feature of Java 1.5.0. The reason for this is that three types are involved in compose, namely:

1. The type of the domain of the inner map.

2. The type of the range of the inner map, which is equal to the type of the domain of the outer map.

3. The type of the range of the outer map.

This cannot be done using the generic feature in Java, which only allows use of the types specified in the signature of the class. Only two types are specified for *RSLMap*, namely the type of the domain and the type of the range of the map. The idea of the compose operator in RSL is to create a new map from the domain of the first map to the range of the second map. An entry in the new map is created if a value in the range of the first map can be found in the domain of the second map. In Java, this is done by creating a new instance of *RSLMapDefault*, which is returned as result. Before returning the *RSLMapDefault* instance there is an iteration over the entry set of the inner map. For each entry in the inner map, it is checked, whether the value of the entry is a key in the outer map (the instance on which the method is applied). If the value of the entry is a key in the outer map, then an entry is added to the result with the key of the entry as key, and the value of applying the value of the entry to the outer map as value.

Example 4.3 compose method in class: *RSLMapDefault*

```
private HashMap<K,V> map;

public RSLMap compose(RSLMap innerMap) {
    RSLMap result = new RSLMapDefault();

    Iterator iterator =
        innerMap.getMap().entrySet().iterator();
    while(iterator.hasNext()) {
        Map.Entry elem = (Map.Entry) iterator.next();

        if(this.map.containsKey((K)elem.getValue())) {
            result.getMap().put(elem.getKey(),
                                (V)this.map.get(innerMap.get(
                                    elem.getKey())));
        }
    }
    return result;
}
```

4.2 First Version of the Translator

The first version of the translator, which translates RSL_0 into Java was implemented entirely by hand in Java. This was needed to get the bootstrapping process started. The translator consists of three modules, which take care of the translation process, and a control module. Two of the data structures, namely the RSL AST and the Java AST, were also implemented by hand. To organize these modules and data structures, a number of packages were created:

translator Package containing the other packages, the translation module, and the control module.

translator.rsllib Package containing library classes for representations of lists, sets, and maps.

translator.lib Package containing visitors for traversing the two AST's and common interfaces and classes for the data structures.

translator.javaast Package containing classes for creating a Java AST.

translator.rslast Package containing classes for creating an RSL AST.

translator.syntacticanalyzer Package containing the grammar file and files generated by ANTLR for the front end of the translator.

The first version of the translator was implemented by hand and the library consist of 9482 lines of code. In addition to this, the lexer and parser generated by ANTLR consist of 3421 lines of code. The source code for the first version of the translator and the library classes can be found in Appendix E.

4.2.1 AST Representation of RSL

The AST representation of RSL was, as stated, implemented by hand in the first version and placed in a package, *translator.rslast*. The package contains a large number of classes, which represent different constructs in RSL. There are a number of correspondences between the syntax rules of RSL presented in [11] and the classes implemented. These correspondences will be listed in the following.

A non–terminal symbol of the grammar is implemented as one or more classes depending on the number of alternatives of the right–hand side of the grammar rule for the non–terminal.

If several non–terminal alternatives exist an abstract class is created, and each of the alternatives is implemented as a class extending the abstract class. An example of such a non–terminal is a type definition in RSL which is shown in Example 4.4.

Example 4.4 Syntax rule *type_def* and corresponding Java implementation.

$$
\begin{aligned}
\mathit{type\_def} ::= \\
&\quad \mathit{sort\_def} \mid \\
&\quad \mathit{variant\_def} \mid \\
&\quad \mathit{short\_record\_def} \mid \\
&\quad \mathit{abbreviation\_def} \mid \\
&\quad \mathit{union\_def}
\end{aligned}
$$

Implemented in the following classes in Java:

```
public abstract class TypeDef {...}
public class SortDef extends TypeDef {...}
public class VariantDef extends TypeDef {...}
public class ShortRecordDef extends TypeDef {...}
```

The rest of the type definitions were not implemented because they are not part of RSL_0.

A right–hand side of a non–terminal symbol which consists of several other symbols is implemented as a class with a number of fields corresponding to the non–terminal symbols of the right–hand side of the rule.

The syntax uses two postfixes on the constructs: *string* and *list*. *string* is a whitespace separated list of a construct, and *list* is a comma separated list of a construct. Both these kind of lists are implemented using the library class provided namely *RSLListDefault*.

An example of such a non–terminal is the *application_expr*. An *application_expr* consist of a *value_expr* and an *optional-value_expr-list* in RSL_0, the rule and the implemention is shown in Example 4.5.

If the right–hand side of a non–terminal consists of only terminal symbols, the non–terminal is implemented as a class with a constant corresponding to each of the terminal symbols. An example of this is the *infix_op* rule, which is shown in Example 4.6.

The RSL AST implements the visitor design pattern as described in Section 3.1.2. More precisely, it implements the variant of the design pattern where the traversal of the structure is placed in the visitor. The source code for the classes implementing the RSL AST can be found in Appendix E.3.

Example 4.5 Implementation of non–terminal containing other non–terminals.

$$\begin{aligned} \mathit{application\_expr} ::= & \\ & \mathit{value\_expr} \\ & \text{optional} - \mathit{value\_expr} - \mathit{list} \end{aligned}$$

Implemented in Java in the following class:

```
public class ApplicationExpr extends ValueExpr implements
    Element {
    private ValueExpr valueExpr;
    private RSLList<ValueExpr> optionalValueExprList;

    public ApplicationExpr(ValueExpr valueExpr,
                           RSLListDefault<ValueExpr>
                              optionalValueExprList) {
        this.valueExpr = valueExpr;
        this.optionalValueExprList = optionalValueExprList;
    }
    ...
}
```

4.2.2 AST Representation of Java

The AST representation of Java was implemented by hand in the first version of the translator. The implementation was based on two sources, namely [13, Appendix L] and [5]. The implementation uses the same principles as the implementation of the RSL AST. A non–terminal symbol whose, right–hand side consists of several non–terminal alternatives is implemented as an abstract class with a number of class extending it. An example of such a construct is the *statement* in Java, which may be a number of different statements. The rule and the implementation is shown in Example 4.7.

If the right–hand side of a non–terminal with one alternative consists of several non–terminals, it is implemented as a class with a number of fields, corresponding to the non–terminal symbols of the right–hand side. An example of this is the *if–statement* in Java, which is shown in Example 4.8.

A right–hand side of a non–terminal consisting of several alternatives, which are all terminal symbols, is implemented as a class containing a constant for each of the alternatives. An example of this is the *modifier* in Java, which is shown in Example 4.9.

The Java AST implements the same variant of the visitor design pattern

Example 4.6 Implementation of non–terminal containing only terminal symbols.

$$
\begin{aligned}
infix\_op ::= & \\
& + \mid \\
& = \mid \\
& \star \mid \\
& \frown
\end{aligned}
$$

Implemented in Java in the following class:

```
public class RSLInfixOp extends RSLOp {
    public static final RSLInfixOp
        RSL_INFIX_OP_PLUS = new RSLInfixOp("+");
    public static final RSLInfixOp
        RSL_INFIX_OP_EQUALS = new RSLInfixOp("=");
    public static final RSLInfixOp
        RSL_INFIX_OP_STAR = new RSLInfixOp("*");
    public static final RSLInfixOp
        RSL_INFIX_OP_HAT = new RSLInfixOp("^");

    /*Private field containing text for printing RSL.*/
    private String text;

    /*Private constructor used only here for creation of
       contants.*/
    private RSLInfixOp(String text) {
        this.text = text;
    }
    ...
}
```

Example 4.7 Non–terminal which consists of several alternatives, which are non–terminals.

$$
\begin{aligned}
Statement ::= & \\
& StatementExpression \mid \\
& IfStatement \mid \\
& ReturnStatement \mid \\
& \vdots
\end{aligned}
$$

Implemented in Java in the following classes:

```
public abstract class Statement {...}
public class ExpressionStatement extends Statement {...}
public class IfStatement extends Statement {...}
public class ReturnStatement extends Statement {...}
⋮
```

Example 4.8 Non–terminal containing several non–terminals.

$$
\begin{aligned}
IfStatement ::= & \\
& Expression \mid \\
& Statement \mid \\
& Statement?
\end{aligned}
$$

Implemented in Java in the following class:

```
public class IfStatement extends Statement implements
    JavaElement {
  private Expression condition;
  private Block ifBlock;
  private Block optionalElseBlock;

  public IfStatement(Expression condition, Block ifBlock, Block
      optionalElseBlock) {
    this.condition = condition;
    this.ifBlock = ifBlock;
    this.optionalElseBlock = optionalElseBlock;
  }
  ...
}
```

Example 4.9 Non–terminal which consists of several alternatives, which are all terminals.

$$
\begin{aligned}
\mathit{Modifier} ::= & \\
& \mathit{public} \mid \\
& \mathit{private} \mid \\
& \mathit{static} \mid \\
& \mathit{abstract} \mid \\
& \vdots
\end{aligned}
$$

Implemented in Java in the following class:

```
public class Modifier implements JavaElement {
    public static final Modifier PUBLIC =
        new Modifier("public");
    public static final Modifier PRIVATE =
        new Modifier("private");
    public static final Modifier STATIC =
        new Modifier("static");
    public static final Modifier ABSTRACT =
        new Modifier("abstract");

    /*Private field containing the text which must be used
      in the writing of Java source code.
    */
    private String text;

    /*Private constructor for creating the constants*/
    private Modifier(String text) {
        this.text = text;
    }
    ...
}
```

as the RSL AST. The source code for the classes implementing the Java AST can be found in Appendix E.4.

4.2.3 Front End

The front end of the translator, i.e. the part transforming an RSL ASCII file to an RSL AST was created using the tool ANTLR. ANTLR creates a lexer and a parser based on a grammar file. The specification of the grammar can be found in Appendix D.1.

The grammar file contains the specifications of both the lexer and the parser. The specification of the lexer and parser are placed in two sections in the file. The file starts by stating a few general options, namely which packages should be imported in the files generated and the implementation language of the lexer and parser. The *translator.rsllib* is imported because it is used in the generation of the RSL AST. The implementation language is of course set to Java.

The specification of the lexer starts by setting a number of options. One of these options is the number of characters of lookahead the lexer should use. This option is set to 2 by the statement `k=2;`, to be able to differentiate between the following "*_" where _ represents any character and "*/" which ends a comment section among other things. The lexer allows only the ASCII set of characters, since it is the ASCII representation of RSL that is parsed by setting the option `charVocabulary='\u0000'..'\u007F';`

By default, ANTLR matches each token against a literal table and changes the token type accordingly. This is turned of by setting the option `testLiterals=false;`. This is done to ensure that keywords can be used as part of an identifier. If this was not done, the name `my_hd` could not be used, because the `hd` part would be recognized as the **hd** operator. The option is turned on for the rules recognizing identifiers which can then be changed to other tokens. Comments and white space are ignored by setting the option `$setType(Token.SKIP);` for the two rules matching comments and white space.

The parser section of the grammar also uses a lookahead of two to be able to differentiate between alternatives of a rule having a common prefix. An example of this is the rule `type_def` which has three alternatives; `sort_def`, `variant_def`, and `short_record_def`. All of the rules for matching the three alternatives start with an `IDENT`, therefore a lookahead of two is needed to distinguish between them.

The solution for ensuring different precedence levels between operators are solved, as usual, in a top down parser. To ensure that * are evaluated before +, there are two rules for the operators at these levels. `infix_expr_pr5` matches either an infix expression using an `infix_operator_pr5` or an `infix_expr_pr4`

expression. `infix_expr_pr4` matches either an infix expression using an `infix_operator_pr4` or an `infix_expr_pr3`. The two operators + and * are matched by `infix_operator_pr4` and `infix_operator_pr5` respectively. The rules are shown in Example 4.10.

The action code of the parser generates an instance of RSLAst if parsing is successful. The lexer and parser generated is used by instantiating the lexer with a *FileReader* of the file to parse as argument, and use the lexer as argument to the constructor of the parser. To obtain the result of the parsing, a method in the parser is called. The method corresponds to the starting rule of the grammar. The use of the lexer and parser is listed in Example 4.11.

4.2.4 Translation Module

The translation module in the first version of the translator was implemented by hand. It is implemented in the class *translator.Translator*. The functionality is implemented as a number of static methods. There is one top level function which takes an RSLAst as parameter and returns a JavaAst. This top level function uses other functions for generating parts of the JavaAst.

The RSLAst provided as argument to the method is not type decorated. The type decoration of the RSLAst is initialized by the translation module. The reason for this is that a specification may contain an *extending class expression*, where the *base class* may be a *scheme instantiation*. A scheme instantiation is a reference to a scheme, which is defined in another place, normally in another file. In this case, the translator must start by translating the class of the scheme instantiation, because it may contain types and values, which are used in the extending class. Before the translator is able to parse the class expression of the scheme instantiation, it must be decorated, and therefore the initialization of the type decoration is placed in the translation module. The type decoration is done by first letting a *ParentVisitor* traverse the RSLAst and then let a *TypeDecorateVisitor* traverse the RSLAst. The idea of the visitors was presented in Section 3.2.3. The source code for the visitors can be found in Appendix E.5.

The creation of the JavaAst is done by the top–level method which creates an instance of a JavaAst and returns it. The contents of the JavaAst are results of invocations of other methods, as shown in Example 4.12.

Many of the constructs in an RSLAst contain a list of some constructs, which must be handled individually. This is done by using recursion in the following way. One method checks, using an if statement, whether the list is empty or not, and if the list is empty, then the recursion stops. In case the list is not empty, then the method invokes a second method on the head

Example 4.10 Rules for different precedence levels of operators.

```
1   infix_expr_pr5 returns [ValueExpr ve] {
        ve = null;
        ValueExpr ve1 = null;
        RSLInfixOp rio = null;
5       ValueExpr ve2 = null;
        ValueExpr ve3 = null;
    }
        :   ve1 = infix_expr_pr4
            (
10              rio = infix_op_pr5 ve2 = infix_expr_pr4
                    {ve3 = ve1; ve1 = new ValueInfixExpr(ve3, rio, ve2);}
            )*
            {ve = ve1;}
        ;
15
    infix_expr_pr4 returns [ValueExpr ve] {
        ve =  null;
        ValueExpr ve1 = null;
        RSLInfixOp rio = null;
20      ValueExpr ve2 = null;
        ValueExpr ve3 = null;
    }
        :   ve1 = disamb_expr
            (
25              rio = infix_op_pr4 ve2 = disamb_expr
                    {ve3 = ve1; ve1 = new ValueInfixExpr(ve3, rio, ve2);}
            )*
            {ve = ve1;}
        ;
30
    infix_op_pr5 returns [RSLInfixOp rio] {rio = null;}
        :   PLUS {rio = RSLInfixOp.RSL_INFIX_OP_PLUS;}
        |   HAT {rio = RSLInfixOp.RSL_INFIX_OP_HAT;}
        ;
35
    infix_op_pr4 returns [RSLInfixOp rio] {rio = null;}
        :   STAR {rio = RSLInfixOp.RSL_INFIX_OP_STAR;}
        ;
```

Example 4.11 Front end of the first version.

```
RSLLexer lexer = new RSLLexer(fr);
RSLParser parser = new RSLParser(lexer);
RSLAst rslast = parser.rslast();
```

Example 4.12 Top level function of translation module.

```
public static JavaAst rslast2javaast(RSLAst rslast) {
    JavaAst _v0 = null;

    RSLMap < String, CompilationUnit > compilationUnitMap =
        rslast2javaast(
            rslast.getLibModule().getSchemeDef().getClassExpr(),
            rslast
        );

    _v0 = new JavaAst(compilationUnitMap);

    return _v0;
}
```

of the list and invokes itself on the tail of the list. This way of recursively running through the elements of a list is also used in a slightly more complex way. The recursion may be combined with a check of whether the head fulfils some condition or not. Only if the condition is met, the second method is invoked. An example of this is shown in Example 4.13 where the method makeClassDeclarationList1 checks, whether a list of declarations is empty or not. If the list of declarations is empty, it returns an empty list, else it checks if the head of the list is a type declaration. If it is a type declaration, then it invokes the method makeClassDeclaration2 on the head of the list before it invokes itself on the tail of the list.

In the first version of the translation module, a number of features may be controlled during translation. These features include, among other things, whether or not the classes generated in the translation of type definitions should implement the visitor design pattern, and in case the design pattern is implemented then the name of the visitor. The method makeVisitorMethod is invoked by the methods translating type definitions. The makeVisitorMethod either returns an empty list of methods or a list containing an accept method for the class. The name of the visitor is retrieved from a property file by the statement translator.getProperty("visitor");. The method is shown in Example 4.14. A description of the properties which may be controlled in the translation can be found in Appendix A.

Example 4.13 Use of recursion in the translation module.

```
public static RSLList < ClassDeclaration >
    makeClassDeclarationList1(RSLList < Decl > dl) {
    RSLList < ClassDeclaration > _v0 = null;
    if (dl.equals(new RSLListDefault < Decl > ())) {
        RSLList < ClassDeclaration > _v1 = null;
        _v1 = new RSLListDefault < ClassDeclaration > ();
        _v0 = _v1;
    }
    else {
        if (dl.hd() instanceof TypeDecl) {
            RSLList < TypeDef > tdl = ( (TypeDecl) dl.hd()).
                getTypeDefList();
            RSLList < ClassDeclaration > _v1 = null;
            _v1 = makeClassDeclarationList2(tdl).concat(
                      makeClassDeclarationList1(dl.tl()));
            _v0 = _v1;
        }
        else {
            RSLList < ClassDeclaration > _v1 = null;
            _v1 = makeClassDeclarationList1(dl.tl());
            _v0 = _v1;
        }
    }
    return _v0;
}
```

Example 4.14 *makeVisitorMethod* method.

```
public static RSLList < MethodDeclaration >
    makeVisitorMethod(Id id, boolean abstractMethod) {
  RSLList < MethodDeclaration > _v0 = null;
  if (createVisitorMethods) {
    if (abstractMethod) {
      _v0 = ...
    }
    else {
      _v0 = new RSLListDefault < MethodDeclaration > (
          new MethodDeclaration(
            new RSLListDefault < Modifier > (Modifier.PUBLIC),
            new SimpleName("accept"),
            PrimitiveType.JAVA_VOID,
            new RSLListDefault < SingleVariableDeclaration >
              (new SingleVariableDeclaration(
                    new RSLListDefault < Modifier > (),
                    new ReferenceType(
                        (translator.getProperty("visitor") !=
                            null ?
                          new SimpleName(
                              translator.getProperty("visitor"
                                  )) :
                          new SimpleName("TYPE_UNKNOWN")),
                              null),
                    new SimpleName("visitor"),
                    null)),
            new Block(new RSLListDefault < Statement >
                    (new ExpressionStatement(
                        new MethodInvocation(
                            new SimpleName("visitor"),
                            makeVisitName(id),
                            new RSLListDefault < Expression
                                >
                            (new ThisExpression(null))))))
          )
        );
    }
  }
  else {
    _v0 = new RSLListDefault < MethodDeclaration > ();
  }
  return _v0;
}
```

In Chapter 2, the exact translations of the different constructs in RSL were treated. The general idea of the translation of a basic class expression is that it should result in a number of classes, according to the discussion in Section 2.3.3. Each class should be placed within its own *CompilationUnit*. Each *CompilationUnit* represents a file in the output of the translator. The basic class expression results in one class named after the scheme containing the basic class expression. This class contains methods corresponding to the functions in the basic class expression. The type definitions in the basic class expression result in a number of classes named after the type definitions, according to the translations in Section 2.3.3. If the basic class expression contains test cases, a main method is generated with a System.out.println statement for each test case.

These three issues are handled by three different methods in the translation module. The result of the methods are placed in three different places in the resulting JavaAst.

The translation of functions in a basic class expression is handled by a method makeMethodDeclarationList1, which runs recursively through the list of declarations given as parameter and builds a list of method declarations in Java. makeMethodDeclarationList1 uses the scheme of recursion presented above, checking the list of declarations for value declarations. For each value declaration, it invokes makeMethodDeclarationList2. makeMethodDeclarationList2 uses the same scheme of recursion checking the value definitions for explicit function definitions. For each explicit function definition, it invokes makeMethodDeclaration. makeMethodDeclaration creates a method declaration and returns it to makeMethodDeclarationList2, which wraps the method declaration in a list and concatenates the list to the result. The block of the method declaration is created by another function makeBlock, which is invoked with the value expression of the explicit function definition as argument. The idea of the method makeBlock was presented in Section 2.4.3. The result of makeMethodDeclarationList1 is a list of method declarations, which can be used as a parameter in the creation of a class in Java corresponding to the scheme being translated.

A basic class expression may contain test cases. The test cases result in a main method in the class created. This is done by a method makeMainMethodDeclaration and a number of auxiliary methods, which run through the list of declarations in the basic class expression looking for test case declarations and test cases inside of them. The methods use the same scheme of recursion as makeMethodDeclarationList1. If a test case declaration and a test case definition is found, makeMainMethodDeclaration returns a list of methods containing one method main, else the method returns an empty list of methods. The list of methods returned by makeMainMethodDeclaration

is concatenated with the list created by makeMethodDeclarationList1.

Type declarations in a basic class expression is handled by a third method makeClassDeclaration1, which, like the methods described above, runs through the list of declarations. For each kind of type definition translated there is an auxiliary method for creating the corresponding classes. The methods inside the classes generated are created by other auxiliary methods. These auxiliary methods run through the content of the type definitions and creates the equals and the toString methods needed according to the translation defined in Chapter 2. The result of makeClassDeclaration1 is a list of *ClassDeclaration*'s.

In the top level method rslast2javaast each class declaration is wrapped in a *CompilationUnit* and the result is joined with a *CompilationUnit* created from the translation of value declarations and test case declarations.

The idea of the methods for translating value expressions is given in Section 2.4.3, which describes three methods for carrying out the translation. In the actual implementation, there are more methods, but the idea remains the same. The difference is that instead of having three large methods: One for creating blocks, one for creating statements, and one for creating expressions, each of these is split into a number of overloaded methods, which handles one kind of value expressions each. The difference is illustrated in Example 4.15.

The methods involved in the translation of value expressions contain, among others, these two parameters VariableDeclarationStatement vds and **int** currentVariableNumber. The purpose of these two parameters are to perform the translation as illustrated in Example 2.34 namely to introduce a new variable for each block entered, assigning the result to the variable, and at the end of the block, assign the new variable to the variable of the enclosing block. The parameter vds holds the variable to which the result of the block must be assigned, and the parameter currentVariableNumber holds the number with which the variable may be named to avoid name clashes with the variables of the enclosing blocks. For each nested level currentVariableNumber is incremented. The variables are named "_v0" at the first level and "_v1" at the second level and so on. The method makeBlock takes a third parameter, which is a boolean value stating whether the last statement should be a return statement instead of an assignment. The source code for the translation module can be found in Appendix E.1.1.

4.2.5 Back End

The back end of the first version of the translator was implemented by hand as a visitor, *StringJavaVisitor*, which simply traverses a JavaAst and writes the corresponding Java source code. The visitor design pattern implemented

Example 4.15 Illustration of difference between implementation and idea presented in section 2.4.3.

```
public static RSLList < Statement >
    makeStatementList(IfExpr ifExpr,
                      VariableDeclarationStatement vds,
                      int currentVariableNumber) {
    ...
}

public static RSLList < Statement >
    makeStatementList(CaseExpr caseExpr,
                      VariableDeclarationStatement vds,
                      int currentVariableNumber) {
    ...
}

...

Instead of

public static RSLList < Statement >
    makeStatementList(ValueExpr valueExpr,
                      VariableDeclarationStatement vds,
                      int currentVariableNumber) {
    if(valueExpr instanceof IfExpr)
        ...
    else if(valueExpr instanceof CaseExpr)
        ...
    ...
}
```

by the JavaAst is the variant with the traversal of the object structure placed in the visitor, this makes it is very to generate the Java source code. It is easy to control where the code of the subparts of a construct should be written.

One example of this is the if statement, which is shown in Example 4.16. The visitor starts by writing the **if** keyword, then it invokes the accept method of the condition with itself as argument for writing the condition. After the condition has been written it invokes the accept method of the if–block for writing this part. Then it checks if there is an optional else block, and if so invokes the accept method of the else–block.

Example 4.16 Visit method for if–else statement.

```
public void visitIfStatement(IfStatement ifStatement) {
    result.append("if(");
    ifStatement.getCondition().accept(this);
    result.append(")\n");
    ifStatement.getIfBlock().accept(this);
    result.append("\n");
    if (ifStatement.getOptionalElseBlock() != null) {
        result.append("else");
        ifStatement.getOptionalElseBlock().accept(this);
        result.append("\n");
    }
}
```

Two properties control whether or not to write the generated Java source code to files or to print them to *stdout*. In case the Java source code is written a property controls to what directory the files are written. The description of the properties can be found in Appendix A. The source code for the *StringJavaVisitor* can be found in Appendix E.5.7.

4.2.6 Control Module

The control module in the first version of the translator is very simple. It is implemented as a main method within the *Translator* class. The source code for the control module may be found in Appendix E.1.1 line 2235. The purpose of this module consists of two parts. The first purpose is to be user interface of the translator. It is the control module which must be executed by a user on a command–line. The second purpose is to initiate and control the translation. The translation process is, as shown in Figure 3.20 on page 89 and summarized below. The names in the summary correspond to the names of the figure.

1. A filestream of the file containing the specification is passed on to the front end, *RSLLexer* and *RSLParser*.

2. The result of the front end is an RSLAst which is stored in a variable.

3. The RSLAst is passed on as argument to the translation module, *Translator*.

4. The result of the translation module is a JavaAst which is stored in a variable.

5. The JavaAst is traversed by a visitor creating Java source code, *StringJavaVisitor*.

4.3 Second Version of the Translator

The second version of the translator is implemented in a different way than the first version, due to the bootstrapping process. The front end and the back end of the second version of the translator are implemented in Java like the first version, and they are actually very similar to the first version. The differences lies within the translation module and the data structures, which in the second version were specified in RSL and translated using the first version.

In Section 3.3, two versions of the translator were mentioned, a first version implemented entirely by hand and a second version, where the translation module was specified in RSL_0 and then translated using the first version. The specification of the translation module should handle the slightly larger subset of RSL, RSL_1. In the implementation process, however, there was an intermediate step. First, a translation module handling RSL_0 was specified and translated into a new executable version of the translator. Afterwards, the specification was extended to handle the slightly larger subset RSL_1, and made executable by the steps described in Section 3.3. This intermediate step is mainly for testing purposes. The specified version of the translation module in the intermediate step should produce the same result as the version implemented by hand.

4.3.1 AST Representation of RSL

In the second version of the translator the RSL AST was defined as a specification in RSL and translated using the first version of the translator. To avoid any confusion on the *classpath* between the two versions, all types in

the second version were prefixed with `"TM_"`. The second version is a little different from the first, because the first version uses constants for representation of terminal symbols, whereas the second version uses classes for representation of all symbols. The reason for this is that a non–terminal with only terminal symbols on the right–hand side is specified as a variant definition with alternatives only consisting of constants. The translation of such a variant definition is, as described in Chapter 2, an abstract class with an empty class extending the abstract class for each alternative of the variant definition. An example of this difference is shown in Example 4.17 where the implementation and specification of the rule for an infix operator in RSL is listed.

4.3.2 AST Representation of Java

The Java AST was specified in RSL in the second version of the translator. As with the RSL AST, all types were prefixed with `"TM_"` to avoid confusion on the *classpath* between the different versions. The Java AST was also changed, like for the RSL AST. Non–terminals containing only terminal symbols are also implemented as a variant definition resulting in the use of classes rather than constants in the Java translated.

4.3.3 Front End

The front end of the second version of translator is very similar to the first version. The second version is also implemented using ANTLR, i.e. it is a grammar file, from which ANTLR generates a lexer and a parser. The grammar file of the second version can be found in Appendix D.2. The rules of the second version are very similar to the rules of the first version. There are two differences between the two versions. The second version must recognize RSL_1 rather than RSL_0. Furthermore, the data structure has changed between the two versions. This means that the action code of the grammar has changed. The second version of the front end must generate an RSL AST corresponding to the one required by the second version of the translation module as described in Section 4.3.1.

4.3.4 Translation Module

Even though the translation module in the second version was specified in RSL and translated using the first version of the translator. The principles of the translation between the two AST representation of the languages remain the same. An overview of the specification is given in Chapter 5. All the

Example 4.17 Difference in implementation and specification of prefix operators in RSL.

$$
\begin{aligned}
prefix\_op ::= \\
& hd \mid \\
& tl
\end{aligned}
$$

Implementation in Java, First version:

```
public class RSLPrefixOp extends RSLOp {
    public static final RSLPrefixOp RSL_PREFIX_OP_HD =
        new RSLPrefixOp("hd");
    public static final RSLPrefixOp RSL_PREFIX_OP_TL =
        new RSLPrefixOp("tl");

    /*Private field containing text needed for printing RSL.*/
    private String text;

    /*Private constructor used only in creation of constans.*/
    private RSLPrefixOp(String text) {
        this.text = text;
    }
    ...
}
```

Specification in RSL, Second version:

TM_PrefixOperator == TM_RSL_HD | TM_RSL_TL

Translated into Java:

```
public abstract class TM_PrefixOperator {...}

public class TM_RSL_HD extends TM_PrefixOperator {...}

public class TM_RSL_TL extends TM_PrefixOperator {...}
```

properties, which in the first version were controlled by a property file, are in the second version changed to parameters to the modules. The reason for this is that the use of property files cannot be specified in the subset of RSL used for the specification. The properties are still kept in a property file in the second version of the translator. The access to the property file is moved to the control module, which reads the properties and passes them on to the other modules as parameters.

The type decoration of the RSL AST is in the second version also implemented as a visitor. The principle of the visitor is the same as in the first version, but the different data structures makes it a little different. One of the differences lies in the fact that the visitor at some points needs to go through a wrapper construct to get to the actual construct. An example of this is an enumerated list expression. To be able to use an enumerated list expression as a type of its own in RSL, it is specified as a short record definition in the data structure. A list expression is implemented as a variant definition with the different possible list constructs as alternatives. The alternative for an enumerated list expression is a record constructor `Make_TM_Enumerated_List_Expr` which contains a `TM_Enumerated_List_Expr` as a component. The type decorate visitor must go through the class `Make_TM_Enumerated_List_Expr` to get to the actual enumerated list expression.

In the construction of the AST representation of Java, there are a lot of similarities between the two versions of the translator. The first version of the translation module was written using recursion and static methods in Java, which corresponds to the Java source code generated, when using explicit function definitions and recursion in RSL in a scheme not instantiated by objects. Many of the functions in the specification of the second version correspond to methods in the implementation of the first version. Three of the main methods described for the first version can actually be found as functions in the specification.

1. The method makeMethodDeclarationList1 in the first version is in the second version the function `makeMethodDeclarationList1`.

2. The method makeMainMethodDeclaration in the first version is in the second version the function `makeMainMethodDeclaration`.

3. The method makeClassDeclarationList1 in the first version is in the second version the function `makeCompilationUnitList1`.

The signatures of the methods and functions are actually identical for the first two items listed. makeMethodDeclarationList1 as method takes a list of

declarations and returns a list of method declarations, this is also the case for the function in the second version. makeMainMethodDeclaration as a method takes a list of declarations and returns a list of method declarations. The same is the case for the function `makeMainMethodDeclaration` in the second version. The method makeClassDeclarationList1 is the only one which has changed. The first change is that the method in the first version only takes a list of declarations as parameter, whereas the function in the second version takes four more parameters, three of the parameters are used to create compilation units rather than classes. These three parameters hold the package declaration, the import statement list, and an optional id if the class must extend another class, which is needed to create a compilation unit. The fourth extra parameter is used to determine, whether the classes created should implement the visitor design pattern, which in the first version was controlled by a property. The idea of the method `makeClassDeclarationList1` and the function `makeCompilationUnitList1` are the same namely to create the classes which make up the translation of the type definitions in a basic class expression.

These correspondences between methods in the implementation of the first version and functions specified in the second version can be found many times in the two versions of the translation module. There may be some variations between the number of parameters for the methods and the functions, but this is mainly due to the fact that the properties must be passed around between the functions, until they are used in some function in the hierarchy of function calls. An example of this is the `TM_Optional_Id, visitorId`, which is a parameter to the top–level function `rslast2javaast`. This parameter indicates whether or not the classes created in the translation of type definitions should implement the visitor design pattern, and if the visitor design pattern is implemented, then the name of the visitor is the `TM_Id` of the `TM_Optional_Id`. The parameter `TM_Optional_Id, visitorId` is part of the functions listed below, which eventually calls the function `makeVisitorMethod` that uses the parameter.

1. `rslAst2JavaAst`
2. `rslAst2CompilationUnitList`
3. `makeCompilationUnitList1`
4. `makeCompilationUnitList2`
5. `makeClassDeclarationList1` which checks whether the method `makeVisitorMethod` should be applied or not, by checking whether `visitorId` is a `Make_TM_Id` or not.

6. `makeVisitorMethod` creates a method based on the `TM_Id` in the `visitorId` .

Another difference between the two versions of the translator is the data structures used in the two versions. The two versions of the data structures are described previously in this chapter and the differences between them have some influence on how the translation module works. An example of the difference is shown in Example 4.18. The Java AST in the second version requires the translation module to wrap the value literal after creation.

Example 4.18 Differences in creation of an if statement between the first and second version of the translator.

Creation of a value literal in the first version:

```
public static Expression makeExpression(ValueExpr valueExpr) {
  Expression _v1 = null;
  ...
  else if (valueExpr instanceof ValueLiteral) {
    Expression _v2 = null;
    ...
    else if (valueExpr instanceof ValueLiteralInteger) {
      ValueLiteralInteger vli = (ValueLiteralInteger)
          valueExpr;
      _v2 = new IntegerLiteral(vli.getText());
    }
    ...
  }
}
```

Creation of a value literal in the second version:

```
makeExpression(ve) ≡
case ve of
  Make_TM_ValueLiteral(vl) →
    case vl of
      TM_ValueLiteralInteger(t) →
        Make_TM_JavaValueLiteral(
          Make_TM_JavaValueLiteralInteger(
            mk_TM_JavaValueLiteralInteger(t))),
```

.

4.3.5 Back End

The back end of the second version of the translator is very similar to the first version. The back end is implemented as a visitor traversing the Java AST and writing the corresponding Java source code. The major difference between the first and the second version of the back end is the differences between the two data structures. The principle of the back end is exactly the same in the two versions.

In the second version of the back end, the use of properties in the translation was moved to the control module. The reason for this was to let only one module be responsible for handling a property file.

4.3.6 Control Module

The control module is a little larger in the second version compared to the first version. In the first version, the control module was implemented as a main method inside the translation module, whereas in the second version it is placed in a separate class. The control module uses file access, which cannot be specified in the subsets of RSL translated. Therefore, it cannot be part of the modules which are specified, hence, it cannot be part of the translation module.

In the second version, the control module is the only module which accesses a property file. The properties are passed on as parameters to the other modules. The principle of the control module is exactly the same as in the first version: it serves as interface for the users and it is responsible for initialization of the other modules. The translation is started by calling the front end and passing on a file stream of the file containing the specification. The result of the front end is stored in a variable and passed on to the translation module. The result of the translation module is stored in another variable and then traversed by the visitor of the back end for generating the output of the second version of the translator. The source code of the control module of the second version can be found in Appendix F.1.1.

Chapter 5

Overview of the Formal Specification

This chapter gives an overview of the formal specification of the translator. The translation module is, as stated in Chapter 3, developed using a bootstrapping process. The bootstrapping process means that the specification of the translation module are defined in a number of versions.

In the overview of the development process given in Section 3.3 it was stated that there are two versions of the translator. One which was implemented by hand, capable translating the subset of RSL, named RSL_0, and a second version of the translator, which was specified in RSL_0 and translated using the first version of the translator. The second version of the translator should then be able to translate the slightly greater subset of RSL named RSL_1.

As stated, in Section 4.3 there are two version of the translator which are specified in RSL:

1. An intermediate version between the first and the second version.

2. The second version.

The intermediate version was specified in RSL_0 and is able to translate RSL_0. This intermediate version was used for translating the second version of the translator.

To avoid confusion between the two versions of the specification, a list of the modules is given here.

First version of the specification:

1. RSLAst_Module

2. JavaAst_Module
3. RSLAst_WrapperModule
4. RSLAst_WrapperModule_2
5. Translation_Module

Second version of the specification:

1. RSLAst_Module2
2. JavaAst_Module2
3. RSLAst_WrapperModule2
4. RSLAst_WrapperModule_22
5. Translation_Module2

The schemes listed above are described in the following sections. All the specifications are specified using the subset of RSL named RSL_0.

5.1 Abstract Syntax Tree for RSL

The specification of the abstract syntax tree is based on the definition of the syntax, which may be found in [11] with the addition of *test cases* added by the people at UNU/IIST [1]. The specification only defines the types necessary to build an abstract syntax tree of a specification in RSL. There are no values defined in this module.

The specification is developed using a few simple ideas. The ideas are based on the structure of the right–hand side of a non–terminal in the grammar. The ideas presented in the following correpond to the ideas used in the implementation of the first version of the translator.

A non–terminal of the following form:

$$\begin{aligned} NT &::= NT_1 \mid \ldots \mid NT_n \\ NT_1 &::= NT_{i_1} \ldots NT_{j_1} \\ &\vdots \\ NT_n &::= NT_{i_n} \ldots NT_{j_n} \end{aligned}$$

is specified using the following constructs:

NT == Make_NT$_1$(nt$_1$: NT$_1$) |
⋮ |
Make_NT$_n$(nt$_n$: NT$_n$),
NT$_1$ :: nt$_{i_1}$: NT$_{i_1}$ ... nt$_{j_1}$: NT$_{j_1}$,
⋮
NT$_1$ :: nt$_{i_n}$: NT$_{i_n}$ ... nt$_{j_n}$: NT$_{j_n}$

An example of this is the TM_ValueExpr and the TM_VariableOrValueName which is shown in Example 5.1

Example 5.1 Implementation of TM_ValueExpr and TM_VariableOrValueName

value_expr ::=
⋮
value_or_variable_name
⋮
value_or_variable_name ::=*id*

Specification:

TM_ValueExpr == Make_TM_ValueOrVariableName(
value_or_variable_name : TM_VariableOrValueName),
TM_VariableOrValueName :: id : TM_Id

Non–terminals with several alternatives, which all are terminal symbols, are specified as variant definitions with only constants.

$$NT ::= T_1 \mid T_2 \mid \ldots \mid T_n$$

is specified as follows:

NT == T$_1$ | T$_2$ | ... | T$_n$

An example of the second type of rule is the rule for prefix operators in RSL. The syntax rule for the prefix operator and the corresponding specification are shown in Example 5.2.

Example 5.2 Rule: *prefix_op* RSL: TM_PrefixOperator.

$$
\begin{aligned}
prefix\_op ::= & \\
& hd \mid \\
& tl
\end{aligned}
$$

TM_PrefixOperator == TM_RSL_HD | TM_RSL_TL

The ideas presented above are not followed at all times. Some grammar rules, which present a hierachy, are simplified by moving constructs up in the hierachy. An example of this is some of the rules for value expressions – the principle is shown in Example 5.3.

5.2 Abstract Syntax Tree for Java

There are several ways to represent Java in an abstract way according to the literature. The way it is represented in this module is combination of the best parts of two representation found in [13] and [5]. The main reason for this combination is a lack of literature dealing with the new version of Java, 1.5, which at the start of this project was going through beta–testing. The top–level structure is TM_JavaAst which consist of a TM_CompilationUnit–list. The elements of TM_CompilationUnit–list represent the files, which is the output of the translation. The idea in the specification is the same as described in the previous section for the AST representation of RSL.

5.3 Wrapper Modules

Two wrapper modules are specified. The purpose of the two modules is to specify some additional types and functions in RSL, which can be used in the *Translation_Module*.

The first module *RSLAst_WrapperModule* and *RSLAst_WrapperModule2* defines a number of types, which are not part of the AST representation of RSL, but which are used for decorating the RSL AST after it has been parsed. The specification contains one variant definition, which holds a number of record constructors, which represent the different types a node can have in the decorated RSL AST.

The second module *RSLAst_WrapperModule_2* and *RSLAst_WrapperModule_22*

Example 5.3 Simplifying a hierachy.

value_expr ::=
    ...
    structured_expr |
    ...

structured_expr ::=
    ...
    case_expr |
    ...

case_expr ::=
    value_expr
    case_branch-list

Translation according to the rules:

```
TM_ValueExpr == ... |
        Make_TM_StructuredExpr(
            structured_expr : TM_StructuredExpr) |
        ⋮

TM_StructuredExpr == ... |
        Make_TM_CaseExpr(case_expr : TM_CaseExpr) |
        ⋮

TM_CaseExpr :: value_expr : TM_ValueExpr
               case_branch_list : TM_Case_Branch*
```

Simplied to:

```
TM_ValueExpr ==
   TM_CaseExpr(value_expr : TM_ValueExpr,
               case_branch_list : TM_Case_Branch*)
```

defines a number of functions in RSL. The idea is to allow use of these functions in the translation module, therefore the translation module extends this module. The idea of *RSLAST_WrapperModule_2* is that it is never translated into Java. The functions, which are specified in the module, are implemented in Java elsewhere in the system. The reason for doing this is to allow for interaction between the parts of a system, which are specified in RSL and translated, and the parts which are implemented in Java. The signature between the functions defined in *RSL_WrapperModule_2* and the methods implemented in Java must be identical.

5.4 Translation Module

The translation module of the translator is specified in a module called *Translator_Module* and *Translator_Module2*. The purpose of the translation module is to do the translation between an RSL AST and a Java AST.

This translation is done by a function named `rslAst2javaAst`. The function has the following signature:

```
rslAst2JavaAst :
                TM_RSLAst × TM_OptionalPackageDeclaration ×
                TM_ImportDeclaration* ×
                TM_OptionalSimpleName ×
                TM_OptionalId × Text* × Bool
                → TM_JavaAst
rslAst2JavaAst(ra, opd, idl, ext, visitorId,
                ignoreList, writeExtensionFiles) ≡ ...
```

`ra` is the abstract representation of RSL which must be translated.
`opd` is contains either a package declaration or nothing determining to which package the generated classes should belong.
`idl` is a list of packages the generated classes must import.
`ext` is a optional identifier of a Java class if all the generated class must extend a common super class.
`visitorId` is an optional identifier of a visitor class in Java if the generated classes must implement the visitor design pattern.
`ignoreList` is a list of schemes which should be parsed but never translated. To ensure the wrapper module functionality described in the previous section.
`writeExtensionFiles` is a boolean value determining, whether the code generated from translating basic class expression of an extending class expression should be written or not.

The function creates a `TM_JavaAst` using the `mk_TM_JavaAst` function defined in by the short record definition `TM_JavaAst` in the *JavaAst_Module*, the parameters to the function are applications of other functions defined in the translation module. Each of them takes care of one part of the translation. The ideas of the top–level functions have already been presented in Section 4.3.4.

The type decoration must be initialized by the translation module, according to the discussion in Section 4.3.4. This means that the translation module must be able to create an instance of the *TM_TypeDecorateRSLAstVisitor* class and invoke the accept method in *TM_RSLAst* with the visitor as parameter. *TM_TypeDecorateRSLAstVisitor* is the second version of the type decorate visitor. The source code for the visitor can be found in Appendix F.2.7. The translation module does not have any knowledge about *TM_TypeDecorateRSLAstVisitor* and nor does it have any knowledge about an accept method, but the specification should still be type checked using the existing type checker. The solution for this consists of two parts. The actual initialization of the type decoration has been moved to an auxiliary function, `Runner_instance_typeDecorate`. The function is specified in the *RSLAst_WrapperModule_2*, and is therefore never translated, but must be specified elsewhere in the system. To be able to call the method in the translation of the translation module, the method must be placed within the class created. This is achieved by a property used in the translation process, which allows for insertion of Java source code at the end of the class created in the translation of a scheme in RSL. By using these properties and features implemented it is possible for a module specified in RSL to interact with parts of a system implemented in Java. It should be noted that these features are not limited to be used in the translation of the translation module, but can also be used when applying the translator in other projects.

Chapter 6

Test

The tool implemented in this project involves a number of different packages, which have been developed using different tools and languages as described in previously. Therefore, testing the tool has been a little different from the usual test in a software development project.

6.1 Purpose

The purpose of testing is to increase the confidence in the correctness of the tool developed. It cannot be proven through tests that there are no errors in the implementation, but tests can make it plausible that the implementation works correctly.

6.2 Test Schemes

All parts of the different versions of the translator have been tested. To what degree and how the testing have been done vary between the different parts of the translator.

The main part of the testing has been done as black box testing, which tests that the functionality of the translator is as expected [18]. The functionality has been tested by translating numerous examples of small specifications, and controlling that the generated Java source code complied to what was expected according to the translations defined in Chapter 2. The translation module was as stated in Chapter 5, specified in an intermediate version capable of translating RSL_0, and then in a second version capable of translating RSL_1. This result in the following translations and test phases:

1. Implementation of the first version by hand.

2. Testing of the first version using numerous examples.
3. Translation of the intermediate version of the translation module specified in RSL_0 using the first version.
4. Testing of the executable version of the intermediate version using numerous examples.
5. Translation of the second version of the translation module specified in RSL_0 using the executable version of the intermediate version.
6. Testing of the second version using numerous examples.

The steps mentioned in the list above, namely that each version was tested separately, and that the last two versions were translated using the previous version and worked as expected, are good indications that the translator works as expected.

A selection of the examples used for testing are listed below. For each example it is described which part of the translation it tests. The specification of the examples as well as the source code generated can be found in Appendix I.

LISTSUM Tests the translation of: Explicit function definitions, use of list operators, if–then–else expressions and arithmetic operators.

LIST Tests the translation of: Variant definitions using constants and record constructors and case expressions.

APPLICATION Tests the translation of: Short record definitions, application expressions and destructors.

The front end and the back end of the translator have been tested separately. The front end has been tested by using small specifications and writing the generated RSL AST to a file. The generated RSL AST was compared with what was expected according to the rules of the syntax of RSL. The back end has been tested by letting it write Java code from examples of Java AST's and controlling that the results were as expected.

The library classes in the *rsllib* package have been tested using structural testing or white box testing [18]. A test class has been written for each the library classes. Each of the test classes tries every method in a library class and writes the output to *stdout*. The results were then compared to the expected output. This kind of tests could has been done using a test framework like JUnit [2]. This have not been done because of the relative small size and number of library classes. The test classes for the library classes can be found in Appendix G.

6.3 Results

The results of the tests of the library classes were as expected and a print of the results can be found in Appendix H.

The results of the translation of the examples mentioned generated the Java source code as expected. They were all able to be compiled with *javacc*, and those examples which included test case declarations also gave the expected output when executed. The results can be found in Appendix I.

Chapter 7

Extensions and Further Work

This chapter provides suggestions for how to apply and expand the developments of this work. In Section 7.1 it is proposed how the developments in this project can be applied in the development of new tools for RSL. Section 7.2 and 7.3 give proposals for how to expand the translation from RSL into Java. Section 7.4 contains suggestions for how to expand the translator developed to translate a larger subset of RSL.

7.1 Creating Tools Transforming RSL

In order to make the translator developed more reuseable, a visitor for creating RSL from an RSL AST has been developed. This visitor was both used for debugging the front end and making future development of tools manipulating RSL easier. A tool for translating between different forms of RSL, e.g. from applicative RSL to imperative RSL, can use the front end for parsing RSL, specify the transformations in RSL and translate them using the translator created in this project and use the RSL visitor for writing the translated RSL.

The developer should define the tool for transformation of RSL as an RSL specification which is shown in Figure 7.1.

The tool can be made executable by the following steps. Translate the specification to Java using the translator. The translation is shown in Figure 7.2. Combine this tool with the front end and the RSL visitor to create a tool translating between different kinds of RSL. The combination is shown in Figure 7.3.

Figure 7.1 Tool translating between two kinds of RSL AST's implemented in RSL

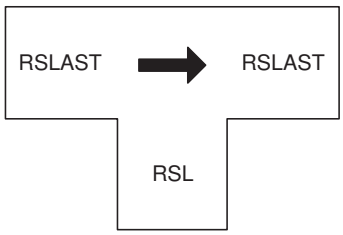


Figure 7.2 Translating the tool specified.

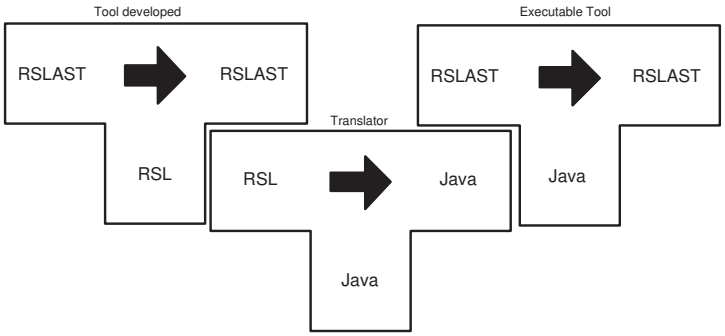
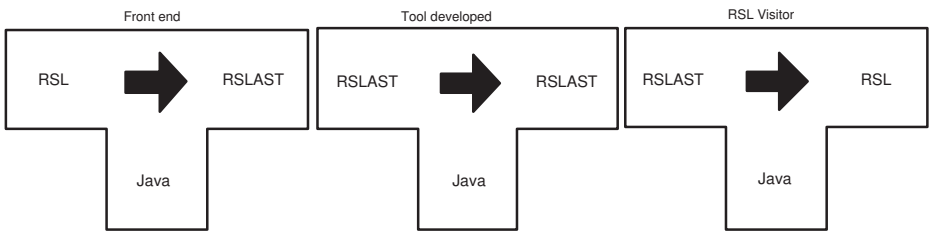


Figure 7.3 Combining the new tool developed with the front end and the RSL visitor as back end.



7.2 Expanding the Translation

The two subsets translated in this project RSL_0 and RSL_1 are very small subsets of RSL. To make the tool more usable the subset translated should be expanded considerable.
Possible steps of extension include:

1. Extend the set of constructs for applicative specification.
2. Add constructs for imperative specification.
3. Add possibilities for specification of communication.
4. Add possibilities for specification of parallel evaluation of expressions.
5. Add possibilities for using axioms and implicit specifications.

The first step of extension should be addition of the translations of constructs for which this project provides a translation, but which were not implemented in the tool. In addition to these the **let** expression and more patterns for the **case** expression should be added, possibly using the translations described in Section 7.3. A final addition to the first step should be the possibility of using pre–condition for functions, a suggestion for implementation of pre–conditions is shown in Section 7.3.

The second step of extension should include possibilities for specifying an imperative system. To be able to specify an imperative system, a number of additions must be made. First of all translation of variable declarations should be added. In addition to this value expressions for assignments to variables as well as value expressions for sequencing should be added. As a final addition to the second step the looping constructs **for**, **while** and **until**, which are very often used in imperative specification, should be added. Possible translations of these constructs are discussed in Section 7.3.

The third step of extension should include possibilities for specifying a system using communication. Communication in RSL is done over channels using input and output value expressions. Therefore, this step of extension must add translation of channel declarations and the value expressions for using channels, namely: input expressions and output expressions.

The fourth step of extension should include possibilities for specifying a system where several expressions may be evaluated in parallel. Parallel evaluation of expressions in RSL is done by using infix combinators. Therefore, this step of extension must include translations of a number of these. $\|$ and $[]$ are the most used infix combinators. Therefore, a translation of these must

be added. The only way processes running in parallel in RSL can communicate is by using channels. The fourth step of extension therefore depends the third step. A discussion of possible translations of the infix combinators is given in Section 7.3.

The final step of extension should include possibilities for specifying a system using axioms. This cannot not be completed entirely, because there are infinitely many ways to define a specification implicitly. The translator must add a handling of each special case of the implicit ways of specification which can be translated. However, the translator can only test for a finite number of these, and therefore this step can never be completed. Even though this step is listed as the last step the handling of some of the special cases may be introduced in any of the other steps.

7.3 Ideas for Translation of Constructs not Treated in Chapter 2

This project includes translations of a number of constructs in RSL, these were treated systematically in Chapter 2. This section contains considerations on translations of some of the constructs not included in Chapter 2. The considerations are not systematically treated, but presents ideas for a translation.

7.3.1 Variable Declarations

"RSL allows declaration of variables as known from programming language like Ada and Pascal. A variable in RSL is a container capable of holding values of a particular type." [11]. Java also allows for declaration of variables which have the same properties. Therefore, a translation of variable declarations in RSL should be as variables in Java. The use of a variable, such as assigning new values to it in RSL, corresponds to assignments in Java. Therefore, an assignment expression in RSL should be translated as an assignment expression in Java. In Java, assignment statements of variables of a reference type are just referenced to object meaning the it is possible to have many variables pointing to the same object. In RSL, on the other hand an assignment expressions assigns a copy of the value to the variable. Therefore, deep–copying of variables of reference types may be needed in Java before making assignments. Sequencing in RSL, where a number of expressions are evaluated in turn, corresponds very much to a block in Java, where a number of statements are executed in turn. For the value expressions sequenced in RSL, only the last expression may be of another type than **Unit**. This

corresponds to a return statement or an assignment statement at the end of the block, according to the use of auxiliary variables in the translator. The use of auxiliary variable was presented Section 2.4.2.

7.3.2 Channel Declarations

Commucation in RSL is specified using a concept called channels. A process may communicate values of the type specified in the **channel** declaration in both directions. In Java, there are a number of posibilities for implementing communication between systems. The possibilities for communication in Java include:

1. Streams.
2. Channels.
3. RMI.
4. CORBA.

Each of these communication concepts in Java have both advantages and disadvantages. However, it must be investigated furher, which of the possible communication schemes provides the best facilities for an implementation in Java.

The most natural choice would be to use channel in Java because they are an implementation of a CSP like channel, which corresponds very much to the channel concept of RSL. There may, however, be some issues with synchronization, because channels in Java are not synchronized in the same way that channels in RSL are. Communication in RSL is at an abstract level therefore there should be defined some ways for determining which system to communicate with and how to reach this system.

7.3.3 Parallel Composition and External Choice

The purpose of the parallel combinator $\|$ in RSL is to allow two expressions to be evaluated concurrently. The two expressions must have the type **Unit**, and it is recommended that they are state–independent. The purpose of the external choice combinator $[]$ in RSL is to specify a choice between different kinds of communication explicitly.

There are no corresponding constructs in Java for these two combinators. However, it is possible in Java to have several threads running in parallel and thereby having several expressions evaluated in parallel.

One suggestion for a translation between value expressions in parallel is to require the value expressions to be application expressions of processes in RSL. The translation of the processes in RSL could then be as classes in Java implementing the *Runnable* interface. The translation of the value expresions of the processes in RSL should be placed in the run method in the Java class. The parallel combinator expression should be kept in another class instantiating the other classes created and starting the execution.

The same solution could be used for dealing with the external choice combinator, namely only allow it to combine processes in RSL and translate them using multiple classes implementing the *Runnable* interface.

7.3.4 Pre–conditions

Explicit functions in RSL may use pre–conditions for testing that some condition is fulfilled before the function is evaluated. This is very useful to ensure that a function does not evaluate to **chaos**. The concept of ensuring that certain conditions are met before evaluating a method would be useful in Java. A pre–condition in RSL consists of a value expression which should be read–only, and it should evaluate to a boolean value. In Java exists a concept, assertions, which has many of the same properties. Assertion statements are used for controlling that an expression is true, if not an *AssertionException* is thrown. Assertion statements can be used inside a method body to check the parameters as well as the state of the program. These similarities indicate that assertions in Java may be a good translation of pre–conditions in RSL. A simple example is a pre–condition ensuring that a function using division does not divide by zero, the example and the translation is shown in Example 7.1.

Example 7.1 Translation of a function using a pre–condition.

value
  partial_fraction : **Real** $\xrightarrow{\sim}$ **Real**
  partial_fration(x) $\equiv$ 1.0/x
  pre x $\neq$ 0.0

```
public static double partial_fraction(double x) {
    assert x != 0.0;
    return 1.0/x;
}
```

7.3.5 Let Expressions

A **let** expression in RSL consists of a number of declarations of values and an expression in which the declared values may be used. One possible translation of this could be as a block in Java. Inside a block in Java, a number of variables may be defined. This corresponds to values declared in a **let** expression. The block should end with the translation of the value expression in the **let** expression. When the execution leaves a block in Java, the variables defined within the block are no longer accessible, which correspond to the fact that names defined in RSL for the values in the **let** declarations can only be used between the keywords **in** and **end**.

7.3.6 Case Expressions

Even though Chapter 2 provides a translation for the **case** expression in RSL, the translation was not complete, because only some of the patterns where translated. The idea of the translation described earlier is good enough, namely that a **case** expression corresponds to a number of if statements in Java, and that the patterns should be translated as conditions of the if statements.

The product pattern could be translated like the record pattern by checking the type using an **instanceof** expression and to define local variables for each part of the product which may be used in the translation of the value expression in the case branch in RSL.

The list pattern comes in two forms. The first form is an enumerated pattern, where each value expression in the list is compared with each value expression in the pattern. The second form the pattern is a concatenated expression. The first form of the list expression could be translated as an infix expression in Java using the `&&` operator. Each element of the list is compared with translation of the corresponding element in the pattern. An example of the translation of an enumerated list pattern is shown in Example 7.2. The translation of the second form of a list pattern is not as simple as the enumerated list pattern. The concatenated list expression contains two patterns which must both match: an enumerated list pattern and an inner pattern, which may be a number of different patterns. The maximal type of the two pattern parts must be equivalent, i.e. they must both be a list of the same type. The first criterion of a match is that each element in the enumerated list pattern must match the first part of the value expression being matched. The second criterion is depending on the type of the second pattern. A wild card has the effect that only the first criterion must match. If the second pattern is a list pattern, then the rest of the list must match

Example 7.2 Idea of translation of list pattern in a case expression.

case l **of**
  ⟨4,5,6⟩ → ...
end

```
if(l.get(1).equals(4) &&
   l.get(2).equals(5) &&
   l.get(3).equals(6)
) {
...
}
```

the list pattern. In case the second pattern is an identifier, then only the first criteria must match, the rest of the list must be declared as a variable, which can be used in the translation of the value expression of the case branch.

An equality pattern should add a criterion to the condition of the if statement. An equality pattern stating that a part of a record pattern or an element of a list must be equal to some value. The translation of an equality pattern should be as shown in Example 7.3.

Example 7.3 Idea of translation of an equality pattern in a case expression.

case hd(cl) **of**
  empty → ...
  add(=White, tail) → ...
  add(=Black, tail) → ...
end

```
if(cl instanceof Empty) {
...
}
else if(cl instanceof Add and cl._v1() instanceof Black) {
...
}
else if(cl instanceof Add and cl._v1() instanceof White) {
...
}
```

7.3.7 For Expressions

A **for** expression in RSL runs through a finite list and evaluates an expression for each member of the list. The syntax is as follows:

for binding **in** value_expr$_1$ **do** value_expr$_2$ **end**

Where the binding may be used in the evaluation of value_expr$_2$.
This looping construct corresponds very much to the new syntax allowed for for–loops in Java 1.5:

```
for(FormalParameter : Expression) Statement
```

Expression must evaluate to an object implementing the interface *java.lang.Iterable*.
FormalParameter corresponds to the binding.
Expression should be the translation of value_expr$_1$.
Statement should be the translation of value_expr$_2$.
An example of the translation is shown in Example 7.4.

Example 7.4 Translation of a for–loop.

```
variable
   result : Real
value
   fraction_sum Int → write result Unit
   fraction_sum(n) ≡
      result := 0.0;
      for i in <1..n> do
         result := result + 1.0/(real i)
      end
```

```
public double result;
public fraction_sum(n) {
    result = 0;
    for(i : (new RSLListDefault(1,n)).getList()) {
        result = result + 1.0/((double) i);
    }
}
```

7.3.8 While Expressions

A while expression in RSL evaluates some value expression repeatedly as long as some other value expresion is evaluated to **true** at the start of each

iteration. This corresponds very much to the idea of a while statement in Java. The syntax of the RSL expression is shown below along with the translation in Java.

while value_expr$_1$ **do** value_expr$_2$ **end**

Where value_expr$_1$ must be of type **Bool**.
Translation in Java:

```
while (ve1) {ve2}
```

ve1 is the translation of value_expr$_1$ as an expression in Java.
ve2 is the translation of value_expr$_2$ as a statement in Java.

7.3.9 Until Expressions

An **until** expression in RSL evaluates some value expression repeatedly until some other value expression is evaluated to **true** at the end of an iteration. The value expression which is to be evaluated repeatedly in an **until** expression in RSL is a least evaluated once, because the test of the conditional value expression is evaluated at the end of the iterations. In Java, there is a do–while construct, which is also evaluated at least once. The difference between the two is that a do–while construct is continued as long as some expression evaluates to true in the end of each iteration, whereas the **until** expression runs until the condition is met. However, the same effect can be obtained in Java by negating the conditional expression. The idea is shown below:

do value_expr$_1$ **until** value_expr$_2$ **end**

Where value_expr$_1$ must be of type **Bool**.
Translation in Java:

```
do {ve1} while(!ve2)
```

ve1 is the translation value_expr$_1$ as a statement in Java.
ve2 is the translation of value_expr$_2$ as expression in Java.

7.4 Translating Larger Subsets of RSL

One of the most valuable extensions of this project would be to extend the translator to translate a larger subset of RSL. This extension would lead to better possibilities for specifying other tools in RSL. It is possible to extend

one part of the translator without extending the rest. The three modules which take care of the translation process may be changed individually. However, if the front end is extended without extending the translation module, some kind of error handling should be added.

My suggestions for the best approach to extend the translator developed in this project are to:

1. Extend the specification of *RSLAst_Module* to a representation of the full RSL.
2. Extend the visitors traversing the RSL Ast to the full RSL.
3. Extend the front end to parse the full RSL.
4. Extend the specification of *JavaAst_Module* to a representation of the full Java language.
5. Extend the visitors traversing the Java Ast to the full Java language.
6. Extend the translation module in a number of steps.

The reason for first extending the specification of the *RSLAst_Module* and the *JavaAst_Module* to the full languages, is that the work then can focus entirely on adding translation of more constructs. The front end and back end would not have to be changed for each of the steps of extension of the translation module. If the full languages are supported in the front end and the back end then they only have to be changed if the languages change.

Ideas for extensions of the translation module were treated in Section 7.3.

Chapter 8

Conclusion

This final chapter summarizes and evaluates the main results of this work.

8.1 Results

The objectives of this project were:

1. To define a *translation* from a subset of RSL into Java, and,
2. to develop a tool, called a *translater*, for translating from RSL into Java according to the defined translation.

These objectives have been fulfilled as discussed in the following. In Section 1.2, some requirements for the translation and the associated translator were stated, each of these requirements and the fulfillment of them will be discussed in the following.
The following was required of the translation:

1. The translation into Java must be semantically equivalent to the specification in RSL.
2. In the Java translated from a specification it should be possible to recognize the specification.

The first requirement of the translation has been fulfilled in the sense that there has been given a discussion of the sensebility of each of the translations defined. The semantics of the two languages have not been defined in formal way that makes it possible to prove the correctness of the translation. Therefore only an informal discussion of the sensibility of the translations has been given. The discussion of the sensibility of the translation was given in Chapter 2.

The recognizability requirement of the translation has been fulfilled in the sense that each translation into Java tries to use constructs in Java which are similar to the constructs in RSL.
The following was required of the translator:

1. The translator should translate a subset of RSL into Java according to the translation defined.

2. The translator should provide extensibility.

The requirement of the translator to translate a subset of RSL into Java according to the translation defined has been fulfilled in the sense that the implementation of the translator is based on the translations defined. The translator has been tested, as described in Chapter 6. The translator is able to translate a number of examples as well as the specification of the translation module. The requirement of the translator to be open for extension is fulfilled in the sense that the design of the translator is modular making it possible to change each of the modules individually. Furthermore, the translation module has been developed using a bootstrapping process opening for a larger subset of RSL to be translated by specifying the new translations in RSL. These possibilities for extending the translator have been shown in Chapter 7.

8.2 Concluding Remarks

This project has been a learning experience for me. The project has given me a better understanding of both the languages. Furthermore, the project has given me a better understanding of the design of a language processor as well as the development of a translator.

The work in this project has shown that it is possible to define a reasonable translation from a subset of the specification language RSL into the programming language Java. Furthermore, the work in this project has shown that even though the subset translated in this project is small, it is strong enough to specify a translation module and thereby also other tools for manipulating languages. The report gives suggestions both for how to extend the subset translated and for how to extend the tool developed.

It is my hope that this project can inspire the development of other tools for RSL and thereby increase the interest for applying formal methods in the development of software.

Bibliography

[1] Univan Ahn and Chris George. C++ translator for the raise specification language. Technical Report 220, UNU/IIST, 2000.

[2] Dave Thomas Andy Hunt. *Pragmatic Unit Testing, in Java with JUnit*. Pragmatic Programmers, 2003.

[3] Unknown author. javacc: Javacc home. available at https://javacc.dev.java.net/.

[4] Thomas H. Cormen. *Introduction to algorithms*. MIT Press, 1989.

[5] Eclipse. Jdt plug-in developer guide. org.eclipse.jdt.core.dom is a package which contains a definition of a Java AST. Part of the documentation when installing a version for further development of Eclipse.

[6] Dick Grune et al. *Modern Compiler Design*. Wiley, 2000.

[7] Erich Gamma et al. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] John R. Levine et al. *lex & yacc*. O'Reilly, 1992.

[9] Étienne Gagnon. Sablecc, an object–oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, 1998.

[10] Chris George. Raise tool user guide. Technical Report 227, UNU/IIST, 2001.

[11] The RAISE Language Group. *The RAISE SPECIFICATION LANGUAGE*. Prentice Hall International (UK) ltd, 1992.

[12] The RAISE Method Group. *The RAISE Development Method*. Prentice Hall International, 1995.

[13] John Lewis and Willam Loftus. *Java software solutions, foundations of program design*. Addison–Wesley, 3rd edition, 2003.

[14] Terrence Parr. Antlr reference manual. available at http://www.antlr.org, filesharing, Recent ANTLR Manual in PDF.

[15] Jesper Gørtz Peter Haff Søren Heilmann Søren Prehn Peter Haastrup Jan Reher Henrik Snog Eld Zierau Peter Michael Bruun, Bent Dandanell. Raise tools reference manual. Technical report, CRI, 1993.

[16] T. Reps. *The Synthesizer Generator Reference Manual*. GrammaTech Inc., 1989.

[17] Friedrich Wilhelm Schröer. The gentle compiler construction system. available at http://www.first.gmd.de/gentle/, 1997.

[18] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[19] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java*. Prentice Hall, 2000.

Appendix A

Using the Translator and Extending the Translator

A.1 Using the Translator

A.1.1 Tools Needed

To use the translator, a number of tools have to be present at the machine.

- Java 1.5.0 or a later edition. More precisely at least J2SE 5.0 or later.
- I have renamed the following commands: java to java1.5 and javac to javac1.5 to be able to have both a J2SE 1.4.* SDK and a J2SE 5.0 in the path.
- An editor for RSL (A this point *emacs* is the obivous choice since the existing tools only work with emacs).
- The type checker for RSL, rsltc. It is not strictly necessary but very handy for controlling that the syntax of RSL is correct before trying to translate it.

A.1.2 Setting up the Translator

Before the translator is ready to be used, a few things needs to be set up. The executable version of the translator has been compiled with J2SE 5.0. If another version of Java is installed on the local computer the source code must be recompiled using the steps listed in Section A.2 without modifying the specifications and source code.

- Copy the executable code of the translator from the provided cd to a directory on the local computer. The executable code for the second version of the translator can be found in the following path: `executable_version_50SE/second_version`.
- Add the directory containing translator to the class path of the system. To avoid having to use the classpath parameter whenever the translator is used.

You should now be ready to try out the first example.

- Copy the specification `LISTSUM.rsl` and `default.properties` from the provided cd to a directory on the local computer. The examples can be found in the directory `examples`.
- Execute the translator on the specification, by the following command in the directory: `java1.5 translator.Runner2 LISTSUM`.
- Compile the translated Java source code: `javac1.5 LISTSUM.java`.
- Execute the compile Java source code: `java1.5 LISTSUM`.

A.1.3 Controlling the Translation Process

The translation is controlled by a property file. The default name of this file is `default.properties` in the current library but another file may be specified as the second argument to the translator. The properties which may be controlled are listed below:

Property Functionality

extensiondir Directory where to look for files which are extended.

writeExtensionFiles Whether or not to write the files generated from translation of base class expression in an extending class expression.

writeFiles Write the result of the translation.

writeFilesDir Directory where the result is written.

createVisitorMethods Whether or not to create visitor methods for the types translated.

visitor The name of the visitor class used in the implementation of the visitor design pattern.

package Inserts a *package* declaration in the top of the classes generated.

import Libraries to import seperated by $.

getMethods Function names which should be translated as dynamic methods, seperated by $.

ignoreList Base classes in RSL not translated, seperated by $.

printRSL Whether or not to print the specification to stdout.

printJava Whether or not to print the translated Java to stdout.

extraMethodFile ASCII file containing extra methods which must be added to the main class in translation.

A.2 Extending the Translator

The directories referenced in this section are relative to the root of the cd provided or the directory to which the content of the cd is copied.

A.2.1 Tools Needed

To extend the translator, the tool mentioned in section A.1.1 must be present as well as the following:

- An editor for Java and ANTLR.
- ANTLR

A.2.2 Extend the Front End

To extend the front end the following steps must be taken:

1. Copy the directories `specification` and `source/second_version` to the local computer.
2. Add constructs to the specification: RSLAst_Module2. Found in directory `specification/second_version`.
3. Translate the module using the `rslast2.properties` as property file. `rslast2.properties` can be found in directory: `specification/second_version`. If the directory: `source/second_version/translator/rslast2` not already exists it must be created.

4. Add RSL constructs which must be recognized to the `rsl2rslast.g` grammar file which can be found in: `source/second_version/translator/syntacticanalyzer`.
5. Generate the new `RSLLexer` and `RSLParser` by using ANTLR: `java1.5 antlr.Tool rsl2rslast.g` in directory: `source/second_version/translator/syntacticanalyzer`.
6. Add visit methods to a least `TM_RSLAstVisitor` and the following visitors if needed:
   (a) `TM_StringRSLAstVisitor.java` to be able to print the RSL.
   (b) `TM_ParentRSLAstVisitor.java` to be able to find the parent construct in the type decoration phase.
   (c) `TM_TypeDecorateRSLAstVisitor.java` if the construct added must be type decorated in some way.

   The visitors can be found in the directory: `source/second_version/translator/lib`.
7. Compile the Java files in the following directories: `source/second_version/translator/lib`, `source/second_version/translator/rslast2`, and `source/second_version/translator/syntacticanalyzer`.

A.2.3 Extend the Translation Module

To extend the translation module, the following steps must be taken:

1. Copy the file `Translator_Module2.rsl` and `translator2.properties` to the `specification/second_version` directory on the local computer (If not already done in extension of the front end).
2. Add or changes functions in the specification Translator_Module2. Translator_Module2 can be found in the directory: `specification/second_version`.
3. Translate the module using the `translator2.properties` as property file, it can be found in directory `specification/second_version`. If the directory `source/second_version/translator` not already exists it must be created.
4. Compile the Java files in the following directory: `source/second_version/translator`.

A.2.4 Extend the Back End

To extend the back end, the following steps must be taken:

1. Copy the files `specification/second_version/JavaAst_Module.rsl`, `specification/second_version/javaast2.properties`, and the directory `source/second_version/translator/lib`. (If not already done in the previous steps).

2. Add constructs to the specification: JavaAst_Module2. JavaAst_Module2 can be found in the directory: `specification/second_version`.

3. Translate the module using the `javaast2.properties` as property file, it can be found in directory `specification/second_version`. If the directory `source/second_version/translator/javaast2` not already exists it must be created.

4. Add visit methods for to both `TM_JavaVisitor` and `TM_StringJavaVisitor` which can be found in `source/second_version/translator/lib`.

5. Compile the Java files in the following directories: `source/second_version/translator/lib` and `source/second_version/translator/javaast2`.

Appendix B

Content of CD

The content of the CD is listed below. Directories are written as `directory`. Content:

`source`

- `first_version` contains source code for the first version of the translator.
- `second_version` contains source code for the second version of the translator.

`executable_version_50SE`

- `first_version` contains precompiled code for the first version of the translator, compiled for J2SE 5.0.
- `second_version` contains precompiled code for the second version of the translator, compiled for J2SE 5.0.

`examples` contains a number of examples and a `default.properties` file.

- `version_2_only` contains an example which only works with the second version of the translator.

`specification`

- `second_version` contains the specification files and property files for the different modules.

Appendix C

Formal Specification

C.1 First Specification, Intermediate Version

C.1.1 Translator_Module

context: RSLAst_WrapperModule_2

```
scheme Translator_Module =
  extend RSLAst_WrapperModule_2 with
  class
    value
      rslAst2JavaAst :
        TM_RSLAst × TM_OptionalPackageDeclaration ×
        TM_ImportDeclaration* ×
        TM_OptionalSimpleName × TM_OptionalId ×
        Text* × Bool →
          TM_JavaAst
      rslAst2JavaAst(
          ra, opd, idl, ext, visitorId, ignoreList,
          writeExtensionFiles) ≡
        mk_TM_JavaAst(
            rslAst2CompilationUnitList(
                class_expr(schemedef(libmodule(ra))), ra, opd,
                idl, ext, visitorId, ignoreList,
                writeExtensionFiles)),

      rslAst2CompilationUnitList :
        TM_ClassExpr × TM_RSLAst ×
        TM_OptionalPackageDeclaration ×
```

```
    TM_ImportDeclaration* ×
    TM_OptionalSimpleName × TM_OptionalId ×
    Text* × Bool →
      TM_CompilationUnit*
rslAst2CompilationUnitList(
    ce, ra, opd, idl, ext, visitorId, ignoreList,
    writeExtensionFiles) ≡
    case ce of
      TM_ExtendingClassExpr(ce1, ce2) →
        rslAst2CompilationUnitList(
            ce1, ra, opd, idl, ext, visitorId,
            ignoreList, writeExtensionFiles) ⌢
        rslAst2CompilationUnitList(
            ce2, ra, opd, idl, ext, visitorId,
            ignoreList, writeExtensionFiles),
      TM_SchemeInstantiation(id) →
        Runner_instance_translate(
            getText(id), writeExtensionFiles),
      TM_BasicClassExpr(dl) →
        if
          textInList(
            getText(id(schemedef(libmodule(ra)))),
            ignoreList)
        then ⟨⟩
        else
          Runner_instance_typeDecorate(ra) ⌢
          ⟨mk_TM_CompilationUnit(
                opd, idl,
                ⟨makeTypeDeclaration(
                      ce, id(schemedef(libmodule(ra))))⟩)⟩ ⌢
          makeCompilationUnitList(
            ce, opd, idl, ext, visitorId)
        end
    end,

makeImportDeclarationList :
    Text* → TM_ImportDeclaration*
makeImportDeclarationList(l) ≡
    if l = ⟨⟩ then ⟨⟩
    else
```

```
        ⟨mk_TM_ImportDeclaration(
              Make_TM_SimpleName(mk_TM_SimpleName(hd l)))⟩ ⌢
        makeImportDeclarationList(tl l)
    end,

makeTypeDeclaration :
    TM_ClassExpr × TM_Id → TM_TypeDeclaration
makeTypeDeclaration(ce, id) ≡
    case ce of
        TM_BasicClassExpr(dl) →
            TM_ClassDeclaration(
                ⟨TM_JAVA_PUBLIC⟩, rslIdToSimpleName(id),
                TM_NoOptionalSimpleName, ⟨⟩, ⟨⟩,
                makeMethodDeclarationList1(dl) ⌢
                makeMainMethodDeclaration(dl), ⟨⟩, ⟨⟩),
        _ →
            TM_ClassDeclaration(
                ⟨⟩,
                mk_TM_SimpleName(
                    "NOT_HANDLED_CLASS_EXPRESSION_RECEIVED"),
                TM_NoOptionalSimpleName, ⟨⟩, ⟨⟩, ⟨
                ⟩, ⟨⟩)
    end,

makeMethodDeclarationList1 :
    TM_Decl* → TM_MethodDeclaration*
makeMethodDeclarationList1(dl) ≡
    if dl = ⟨⟩ then ⟨⟩
    else
        case hd dl of
            TM_ValueDecl(vdl) →
                makeMethodDeclarationList2(vdl) ⌢
                makeMethodDeclarationList1(tl dl),
            _ → makeMethodDeclarationList1(tl dl)
        end
    end,

makeMethodDeclarationList2 :
    TM_ValueDef* → TM_MethodDeclaration*
makeMethodDeclarationList2(vdl) ≡
    if vdl = ⟨⟩ then ⟨⟩
```

```
    else
      case hd vdl of
        TM_ExplicitFunctionDef(st, ffa, ve) →
          ⟨makeMethodDeclaration(st, ffa, ve)⟩ ⌢
          makeMethodDeclarationList2(tl vdl),
        _ → makeMethodDeclarationList2(tl vdl)
      end
    end,

makeMethodDeclaration :
    TM_SingleTyping × TM_FormalFunctionApplication ×
    TM_ValueExpr →
      TM_MethodDeclaration
makeMethodDeclaration(st, ffa, ve) ≡
    mk_TM_MethodDeclaration(
        ⟨TM_JAVA_PUBLIC, TM_JAVA_STATIC⟩,
        makeMethodName(ffa), makeReturnType(st),
        makeArgumentList(ffa, st),
        Make_TM_OptionalBlock(
            makeBlock(
                ve,
                makeReturnVariable(makeReturnType(st), 0),
                true, 0))),

makeMethodName :
    TM_FormalFunctionApplication → TM_SimpleName
makeMethodName(ffa) ≡
    case ffa of
      TM_IdApplication(id, ffpList) →
        rslIdToSimpleName(id)
    end,

makeReturnType : TM_SingleTyping → TM_JavaType
makeReturnType(st) ≡
    case type_expr(st) of
      TM_FunctionTypeExpr(tea, fa, ter) →
        matchType(ter, false, true)
    end,

makeReturnVariable :
    TM_JavaType × Int → TM_VariableDeclarationStatement
```

```
makeReturnVariable(jt, currentVariableNumber) ≡
  case jt of
    Make_TM_PrimitiveType(pt) →
      case pt of
        TM_JAVA_INT →
          mk_TM_VariableDeclarationStatement(
            ⟨⟩, jt,
            mk_TM_VariableDeclarationFragment(
              mk_TM_SimpleName(
                "_v" ⌢
                convert(currentVariableNumber)),
              Make_TM_OptionalExpression(
                Make_TM_JavaValueLiteral(
                  Make_TM_JavaValueLiteralInteger(
                    mk_TM_JavaValueLiteralInteger(
                      "0")))))),
        TM_JAVA_BOOL →
          mk_TM_VariableDeclarationStatement(
            ⟨⟩, jt,
            mk_TM_VariableDeclarationFragment(
              mk_TM_SimpleName(
                "_v" ⌢
                convert(currentVariableNumber)),
              Make_TM_OptionalExpression(
                Make_TM_JavaValueLiteral(
                  Make_TM_JavaValueLiteralBool(
                    mk_TM_JavaValueLiteralBool(
                      "false")))))),
        TM_JAVA_DOUBLE →
          mk_TM_VariableDeclarationStatement(
            ⟨⟩, jt,
            mk_TM_VariableDeclarationFragment(
              mk_TM_SimpleName(
                "_v" ⌢
                convert(currentVariableNumber)),
              Make_TM_OptionalExpression(
                Make_TM_JavaValueLiteral(
                  Make_TM_JavaValueLiteralDouble(
                    mk_TM_JavaValueLiteralDouble(
                      "0.0d")))))),
        TM_JAVA_CHAR →
```

```
                    mk_TM_VariableDeclarationStatement(
                        ⟨⟩, jt,
                        mk_TM_VariableDeclarationFragment(
                            mk_TM_SimpleName(
                                "_v" ⌢
                                convert(currentVariableNumber)),
                            Make_TM_OptionalExpression(
                                Make_TM_JavaValueLiteral(
                                  Make_TM_JavaValueLiteralChar(
                                    mk_TM_JavaValueLiteralChar(
                                      " "))))))),
                _ →
                    mk_TM_VariableDeclarationStatement(
                        ⟨⟩, jt,
                        mk_TM_VariableDeclarationFragment(
                            mk_TM_SimpleName(
                                "_v" ⌢
                                convert(currentVariableNumber)),
                            Make_TM_OptionalExpression(
                                Make_TM_JavaValueLiteral(
                                  Make_TM_JavaValueLiteralString(
                                    mk_TM_JavaValueLiteralString(
                                      "UNKNOWN PRIMITIVE TYPE")))
                            )))
            end,
        Make_TM_ReferenceType(rt) →
            mk_TM_VariableDeclarationStatement(
                ⟨⟩, jt,
                mk_TM_VariableDeclarationFragment(
                    mk_TM_SimpleName(
                        "_v" ⌢ convert(currentVariableNumber)),
                    Make_TM_OptionalExpression(
                        Make_TM_JavaValueLiteral(TM_NullLiteral)
                    )))
    end,

makeArgumentList :
    TM_FormalFunctionApplication × TM_SingleTyping →
        TM_SingleVariableDeclaration*
makeArgumentList(ffa, st) ≡
    case ffa of
```

```
        TM_IdApplication(id, ffpList) →
          case type_expr(st) of
            TM_FunctionTypeExpr(tea, fa, ter) →
              case tea of
                TM_TypeExprList(tel) →
                  ⟨mk_TM_SingleVariableDeclaration(
                        ⟨⟩, matchType(tea, false, true),
                        rslIdToSimpleName(
                          id(hd binding_list(hd ffpList))),
                        TM_NoOptionalExpression)⟩,
                TM_TypeLiteral(tpl) →
                  case tpl of
                    TM_RSL_UNIT → ⟨⟩,
                    _ →
                      ⟨mk_TM_SingleVariableDeclaration(
                          ⟨⟩,
                          matchType(tea, false, true),
                          rslIdToSimpleName(
                            id(hd binding_list(
                                  hd ffpList))),
                          TM_NoOptionalExpression)⟩
                  end,
                TM_TypeExprProduct(te_list) →
                  makeArgumentListTypeExprProduct(
                      binding_list(hd ffpList), te_list),
                TM_TypeName(tn) →
                  ⟨mk_TM_SingleVariableDeclaration(
                        ⟨⟩, matchType(tea, false, true),
                        rslIdToSimpleName(
                          id(hd binding_list(hd ffpList))),
                        TM_NoOptionalExpression)⟩
              end
          end
      end,

    makeArgumentListTypeExprProduct :
      TM_Binding* × TM_TypeExpr* →
        TM_SingleVariableDeclaration*
    makeArgumentListTypeExprProduct(bl, tel) ≡
      if bl = ⟨⟩ then ⟨⟩
      else
```

```
        ⟨mk_TM_SingleVariableDeclaration(
                ⟨⟩, matchType(hd tel, false, true),
                rslIdToSimpleName(id(hd bl)),
                TM_NoOptionalExpression)⟩ ⌢
        makeArgumentListTypeExprProduct(tl bl, tl tel)
    end,

makeArgumentList :
    TM_ComponentKind* × Int →
        TM_SingleVariableDeclaration*
makeArgumentList(ckl, i) ≡
    if ckl = ⟨⟩ then ⟨⟩
    else
        ⟨mk_TM_SingleVariableDeclaration(
                ⟨⟩,
                matchType(type_expr(hd ckl), false, true),
                mk_TM_SimpleName("_v" ⌢ convert(i)),
                TM_NoOptionalExpression)⟩ ⌢
        makeArgumentList(tl ckl, i + 1)
    end,

makeBlock :
    TM_ValueExpr × TM_VariableDeclarationStatement ×
    Bool × Int →
        TM_Block
makeBlock(ve, vds, returnValue, currentVariableNumber) ≡
    if returnValue
    then
        mk_TM_Block(
            ⟨Make_TM_VariableDeclarationStatement(vds)⟩ ⌢
            makeStatementList(
                    ve,
                    makeReturnVariable(
                        getType(vds), currentVariableNumber),
                    currentVariableNumber) ⌢
            ⟨TM_ReturnStatement(
                    Make_TM_JavaName(
                        Make_TM_SimpleName(name(fragment(vds))))
                )⟩)
    else
        mk_TM_Block(
```

```
            ⟨Make_TM_VariableDeclarationStatement(
                    makeReturnVariable(
                        getType(vds), currentVariableNumber))⟩ ⌢
            makeStatementList(
                ve,
                makeReturnVariable(
                    getType(vds), currentVariableNumber),
                currentVariableNumber) ⌢
            ⟨TM_ExpressionStatement(
                    TM_AssignmentExpression(
                        Make_TM_JavaName(
                            Make_TM_SimpleName(
                                name(fragment(vds)))),
                        TM_JAVA_ASSIGNMENT_OP_EQUAL,
                        Make_TM_JavaName(
                            Make_TM_SimpleName(
                                name(
                                    fragment(
                                        makeReturnVariable(
                                            getType(vds),
                                            currentVariableNumber))))
                        )))))
    end,

makeStatementList :
    TM_ValueExpr × TM_VariableDeclarationStatement ×
    Int →
        TM_Statement*
makeStatementList(
     valueExpr, vds, currentVariableNumber) ≡
    case valueExpr of
        Make_TM_IfExpr(ifExpr) →
            makeStatementList(
                ifExpr, vds, currentVariableNumber),
        TM_CaseExpr(c, cbl) →
            makeStatementListCaseExpr(
                c, cbl, vds, currentVariableNumber),
        /*TM_CaseExpr(c, cbl) → ⟨⟩,*/
        _ →
            ⟨TM_ExpressionStatement(
                    TM_AssignmentExpression(
```

```
                    Make_TM_JavaName(
                        Make_TM_SimpleName(
                            name(fragment(vds)))),
                    TM_JAVA_ASSIGNMENT_OP_EQUAL,
                    makeExpression(valueExpr))⟩
    end,

makeStatementList :
    TM_IfExpr × TM_VariableDeclarationStatement × Int →
        TM_Statement*
makeStatementList(ifExpr, vds, currentVariableNumber) ≡
    ⟨TM_IfStatement(
            makeExpression(condition(ifExpr)),
            makeBlock(
                if_case(ifExpr), vds, false,
                currentVariableNumber + 1),
            Make_TM_OptionalBlock(
                makeBlock(
                    else_case(ifExpr), vds, false,
                    currentVariableNumber + 1)))⟩,

makeStatementListCaseExpr :
    TM_ValueExpr × TM_CaseBranch* ×
    TM_VariableDeclarationStatement × Int →
        TM_Statement*
makeStatementListCaseExpr(
     c, cbl, vds, currentVariableNumber) ≡
    makeStatementListCaseExpr(
        cbl, makeExpression(c), vds,
        currentVariableNumber + 1, TM_NoOptionalStatement),

makeStatementListCaseExpr :
    TM_CaseBranch* × TM_Expression ×
    TM_VariableDeclarationStatement × Int ×
    TM_OptionalStatement →
        TM_Statement*
makeStatementListCaseExpr(
     cbl, e, vds, currentVariableNumber, os) ≡
    if cbl = ⟨⟩
    then
        case os of
```

```
            Make_TM_Statement(s) → ⟨s⟩,
            TM_NoOptionalStatement → ⟨⟩
        end
    else
        makeStatementListCaseExpr(
            tl cbl, e, vds, currentVariableNumber,
            Make_TM_Statement(
                makeStatement(
                    hd cbl, e, vds, currentVariableNumber, os)
            ))
    end,

makeStatement :
    TM_CaseBranch × TM_Expression ×
    TM_VariableDeclarationStatement × Int ×
    TM_OptionalStatement →
        TM_Statement
makeStatement(cb, e, vds, currentVariableNumber, os) ≡
    case pattern(cb) of
        TM_NamePattern(id, vi) →
            case os of
                Make_TM_Statement(s) →
                    case s of
                        TM_IfStatement(
                            condition, if_block, else_block) →
                            case else_block of
                                TM_NoOptionalBlock →
                                    TM_IfStatement(
                                        condition, if_block,
                                        Make_TM_OptionalBlock(
                                            mk_TM_Block(
                                                ⟨TM_IfStatement(
                                                    TM_InstanceOfExpression(
                                                        e,
                                                        Make_TM_ReferenceType(
                                                            mk_TM_ReferenceType
                                                            (
                                                    Make_TM_SimpleName(
                                                        rslIdToSimpleName(
                                                            id)),
                                                    TM_NoOptionalReferenceType))),
```

```
                                    makeBlock
                                    (
                                value_expr(cb),
                                vds, false,
                                currentVariableNumber),
                                    TM_NoOptionalBlock)⟩)))
                            ),
                        Make_TM_OptionalBlock(b) →
                            TM_IfStatement(
                              condition, if_block,
                              Make_TM_OptionalBlock(
                                mk_TM_Block(
                                  ⟨makeStatement(
                                    cb, e, vds,
                                    currentVariableNumber,
                                    Make_TM_Statement(
                                      hd statementList(b))
                                  )⟩⟩⟩⟩)
                        end,
                    _ →
                        TM_ExpressionStatement(
                            TM_ClassInstanceCreation(
                                TM_NoOptionalExpression,
                                mk_TM_ReferenceType(
                                  Make_TM_SimpleName(
                                    mk_TM_SimpleName("String")),
                                  TM_NoOptionalReferenceType),
                                ⟨Make_TM_JavaValueLiteral(
                                    Make_TM_JavaValueLiteralString(
                                      mk_TM_JavaValueLiteralString
                                      (
"SOMETHING IS REALLY WRONG")))))))
                end,
            TM_NoOptionalStatement →
                TM_IfStatement(
                    TM_InstanceOfExpression(
                        e,
                        Make_TM_ReferenceType(
                            mk_TM_ReferenceType(
                              Make_TM_SimpleName(
                                rslIdToSimpleName(id)),
```

```
                                TM_NoOptionalReferenceType))),
                    makeBlock(
                        value_expr(cb), vds, false,
                        currentVariableNumber),
                    TM_NoOptionalBlock)
        end,
    TM_RecordPattern(vovn, ipl) →
        case os of
            Make_TM_Statement(s) →
                case s of
                    TM_IfStatement(
                        condition, if_block, else_block) →
                        case else_block of
                            TM_NoOptionalBlock →
                                TM_IfStatement(
                                    condition, if_block,
                                    Make_TM_OptionalBlock(
                                        mk_TM_Block(
                                            ⟨TM_IfStatement(
                                                TM_InstanceOfExpression(
                                                    e,
                                                    Make_TM_ReferenceType(
                                                        mk_TM_ReferenceType
                                                        (
                                    Make_TM_SimpleName(
                                        rslIdToSimpleName(
                                            id(vovn)
                                        )),
                                    TM_NoOptionalReferenceType))),
                                                mk_TM_Block(
                                                    makeStatementList(
                                                        e, pattern(cb),
                                                        ipl) ⌢
                                                    statementList(
                                                        makeBlock
                                                        (
                                                value_expr(cb),
                                                vds, false,
                                                currentVariableNumber))
                                                ), TM_NoOptionalBlock)
                                            ⟩)))),
```

```
                    Make_TM_OptionalBlock(b) →
                      TM_IfStatement(
                        condition, if_block,
                        Make_TM_OptionalBlock(
                          mk_TM_Block(
                            ⟨makeStatement(
                              cb, e, vds,
                              currentVariableNumber,
                              Make_TM_Statement(
                                hd statementList(b))
                            )⟩⟩⟩⟩)
                  end,
                _ →
                  TM_ExpressionStatement(
                    TM_ClassInstanceCreation(
                      TM_NoOptionalExpression,
                      mk_TM_ReferenceType(
                        Make_TM_SimpleName(
                          mk_TM_SimpleName("String")),
                        TM_NoOptionalReferenceType),
                      ⟨Make_TM_JavaValueLiteral(
                        Make_TM_JavaValueLiteralString(
                          mk_TM_JavaValueLiteralString
                          (
"SOMETHING IS REALLY WRONG")))))))
            end,
          TM_NoOptionalStatement →
            TM_IfStatement(
              TM_InstanceOfExpression(
                e,
                Make_TM_ReferenceType(
                  mk_TM_ReferenceType(
                    Make_TM_SimpleName(
                      rslIdToSimpleName(id(vovn))),
                    TM_NoOptionalReferenceType))),
              mk_TM_Block(
                makeStatementList(e, pattern(cb), ipl) ⌢
                statementList(
                  makeBlock(
                    value_expr(cb), vds, false,
                    currentVariableNumber))),
```

```
                TM_NoOptionalBlock)
    end,
  TM_ValueLiteralPattern(vl) →
    case os of
      Make_TM_Statement(s) →
        case s of
          TM_IfStatement(
              condition, if_block, else_block) →
            case else_block of
              TM_NoOptionalBlock →
                TM_IfStatement(
                  condition, if_block,
                  Make_TM_OptionalBlock(
                    mk_TM_Block(
                      ⟨TM_IfStatement(
                          TM_InfixExpression(
                            e, TM_JAVA_EQUALS,
                            makeExpression(
                              Make_TM_ValueLiteral(
                                vl))),
                          makeBlock
                          (
                        value_expr(cb),
                        vds, false,
                        currentVariableNumber),
                          TM_NoOptionalBlock)⟩)))
                ),
              Make_TM_OptionalBlock(b) →
                TM_IfStatement(
                  condition, if_block,
                  Make_TM_OptionalBlock(
                    mk_TM_Block(
                      ⟨makeStatement(
                          cb, e, vds,
                          currentVariableNumber,
                          Make_TM_Statement(
                            hd statementList(b))
                        )⟩)))
            end,
          _ →
            TM_ExpressionStatement(
```

```
                                TM_ClassInstanceCreation(
                                    TM_NoOptionalExpression,
                                    mk_TM_ReferenceType(
                                      Make_TM_SimpleName(
                                        mk_TM_SimpleName("String")),
                                      TM_NoOptionalReferenceType),
                                    ⟨Make_TM_JavaValueLiteral(
                                        Make_TM_JavaValueLiteralString(
                                          mk_TM_JavaValueLiteralString
                                          (
"SOMETHING IS REALLY WRONG")))))⟩)
                end,
            TM_NoOptionalStatement →
                TM_IfStatement(
                    TM_InfixExpression(
                        e, TM_JAVA_EQUALS,
                        makeExpression(
                            Make_TM_ValueLiteral(vl))),
                    makeBlock(
                        value_expr(cb), vds, false,
                        currentVariableNumber),
                    TM_NoOptionalBlock)
        end,
      TM_WildcardPattern →
        case os of
            Make_TM_Statement(s) →
                case s of
                    TM_IfStatement(
                        condition, if_block, else_block) →
                        case else_block of
                            TM_NoOptionalBlock →
                                TM_IfStatement(
                                  condition, if_block,
                                  Make_TM_OptionalBlock(
                                    makeBlock(
                                      value_expr(cb), vds, false,
                                      currentVariableNumber))),
                            Make_TM_OptionalBlock(b) →
                                TM_IfStatement(
                                  condition, if_block,
                                  Make_TM_OptionalBlock(
```

```
                                    mk_TM_Block(
                                      ⟨makeStatement(
                                          cb, e, vds,
                                          currentVariableNumber,
                                          Make_TM_Statement(
                                            hd statementList(b))
                                      )⟩)))
                    end,
                _ →
                    TM_ExpressionStatement(
                        TM_ClassInstanceCreation(
                            TM_NoOptionalExpression,
                            mk_TM_ReferenceType(
                              Make_TM_SimpleName(
                                mk_TM_SimpleName("String")),
                              TM_NoOptionalReferenceType),
                            ⟨Make_TM_JavaValueLiteral(
                                Make_TM_JavaValueLiteralString(
                                  mk_TM_JavaValueLiteralString
                                  (
"SOMETHING IS REALLY WRONG")))⟩))
            end,
        TM_NoOptionalStatement →
            TM_ExpressionStatement(
                makeExpression(value_expr(cb)))
    end,
  _ →
    TM_ExpressionStatement(
        TM_ClassInstanceCreation(
            TM_NoOptionalExpression,
            mk_TM_ReferenceType(
                Make_TM_SimpleName(
                    mk_TM_SimpleName("String")),
                TM_NoOptionalReferenceType),
            ⟨Make_TM_JavaValueLiteral(
                Make_TM_JavaValueLiteralString(
                    mk_TM_JavaValueLiteralString(
                      "NOT_NAMEPATTERN")))⟩))
end,

makeStatementList :
```

```
    TM_Expression × TM_Pattern × TM_Pattern* →
      TM_Statement*
  makeStatementList(
    expression, recordPattern, innerPatternList) ≡
    case recordPattern of
      TM_RecordPattern(vovn, il) →
        if innerPatternList = ⟨⟩ then ⟨⟩
        else
          ⟨Make_TM_VariableDeclarationStatement(
              mk_TM_VariableDeclarationStatement(
                ⟨⟩,
                matchType(
                  getTypeEvaluatorForTypeEvaluation(
                    hd innerPatternList), false, true
                ),
                mk_TM_VariableDeclarationFragment(
                  rslIdToSimpleName(
                    hd innerPatternList),
                  Make_TM_OptionalExpression(
                    TM_MethodInvocation(
                      Make_TM_OptionalExpression(
                        TM_ParenthesizedExpression(
                          TM_CastExpression(
                            Make_TM_ReferenceType(
                              mk_TM_ReferenceType
                              (
                    Make_TM_SimpleName(
                      rslIdToSimpleName(
                        id(vovn))
                    ),
                    TM_NoOptionalReferenceType)),
                              expression))),
                      valueInitializerToSimpleName(
                        hd innerPatternList), ⟨⟩))
                  ))))⟩ ⌢
            makeStatementList(
              expression, recordPattern,
              tl innerPatternList)
        end,
      _ → ⟨⟩
    end,
```

```
/*
makeExpression : TM_ValueExpr → TM_Expression
makeExpression(ve) is
Make_TM_JavaValueLiteral(Make_TM_JavaValueLiteralInteger(
mk_TM_JavaValueLiteralInteger("UNKNOWN_EXPRESSION")
)),
*/
makeExpression : TM_ValueExpr → TM_Expression
makeExpression(ve) ≡
  case ve of
    Make_TM_ValueLiteral(vl) →
      case vl of
        TM_ValueLiteralInteger(t) →
          Make_TM_JavaValueLiteral(
            Make_TM_JavaValueLiteralInteger(
              mk_TM_JavaValueLiteralInteger(t))),
        TM_ValueLiteralReal(t) →
          Make_TM_JavaValueLiteral(
            Make_TM_JavaValueLiteralDouble(
              mk_TM_JavaValueLiteralDouble(t))),
        TM_ValueLiteralBool(t) →
          Make_TM_JavaValueLiteral(
            Make_TM_JavaValueLiteralBool(
              mk_TM_JavaValueLiteralBool(t))),
        TM_ValueLiteralText(t) →
          Make_TM_JavaValueLiteral(
            Make_TM_JavaValueLiteralString(
              mk_TM_JavaValueLiteralString(t))),
        TM_ValueLiteralChar(t) →
          Make_TM_JavaValueLiteral(
            Make_TM_JavaValueLiteralChar(
              mk_TM_JavaValueLiteralChar(t)))
      end,
    TM_ValueInfixExpr(lo, operator, ro) →
      case operator of
        TM_RSL_PLUS →
          TM_InfixExpression(
            makeExpression(lo), TM_JAVA_PLUS,
            makeExpression(ro)),
        TM_RSL_STAR →
```

```
        TM_InfixExpression(
            makeExpression(lo), TM_JAVA_STAR,
            makeExpression(ro)),
        /*
        TM_RSL_SLASH →
        TM_InfixExpression(
        makeExpression(lo),
        TM_JAVA_DIV,
        makeExpression(ro)
        ),
        */
        TM_RSL_HAT →
            TM_MethodInvocation(
                Make_TM_OptionalExpression(
                    makeExpression(lo)),
                mk_TM_SimpleName(″concat″),
                ⟨makeExpression(ro)⟩),
        TM_RSL_EQUAL →
            case
                matchType(
                    getTypeEvaluatorForTypeEvaluation(lo),
                    false, false)
            of
                Make_TM_PrimitiveType(pt) →
                    TM_InfixExpression(
                        makeExpression(lo), TM_JAVA_EQUALS,
                        makeExpression(ro)),
                Make_TM_ReferenceType(rt) →
                    TM_MethodInvocation(
                        Make_TM_OptionalExpression(
                            makeExpression(lo)),
                        mk_TM_SimpleName(″equals″),
                        ⟨makeExpression(ro)⟩)
            end,
        _ →
            Make_TM_JavaValueLiteral(
                Make_TM_JavaValueLiteralInteger(
                    mk_TM_JavaValueLiteralInteger(
                        ″UNKNOWN_OPERATOR″)))
    end,
TM_ValuePrefixExpr(po, value_expr) →
```

```
        case po of
          TM_RSL_HD →
            TM_MethodInvocation(
                Make_TM_OptionalExpression(
                    makeExpression(value_expr)),
                mk_TM_SimpleName("hd"), ⟨⟩),
          TM_RSL_TL →
            TM_MethodInvocation(
                Make_TM_OptionalExpression(
                    makeExpression(value_expr)),
                mk_TM_SimpleName("tl"), ⟨⟩),
          _ →
            Make_TM_JavaName(
                Make_TM_SimpleName(
                    mk_TM_SimpleName("UNKNOWN_OPERATOR")))
        end,
      TM_ApplicationExpr(value_expr, vel) →
        case
          getTypeEvaluatorForTypeEvaluation(value_expr)
        of
          TM_TypeEvaluator_TM_TypeExpr(te) →
            case te of
              TM_TypeName(tn) →
                TM_ClassInstanceCreation(
                    TM_NoOptionalExpression,
                    mk_TM_ReferenceType(
                        Make_TM_SimpleName(
                          rslIdToSimpleName(tn)),
                        TM_NoOptionalReferenceType),
                    makeArgumentList(vel)),
              _ →
                TM_MethodInvocation(
                    TM_NoOptionalExpression,
                    makeMethodName(value_expr),
                    makeArgumentList(vel))
            end,
          TM_TypeEvaluator_TM_Constructor(c) →
            TM_ClassInstanceCreation(
                TM_NoOptionalExpression,
                mk_TM_ReferenceType(
                    Make_TM_SimpleName(
```

```
                        rslIdToSimpleName(id(c))),
                    TM_NoOptionalReferenceType),
                makeArgumentList(vel)),
            TM_TypeEvaluator_TM_Destructor(d) →
                TM_MethodInvocation(
                    Make_TM_OptionalExpression(
                        TM_ParenthesizedExpression(
                            makeExpression(hd vel))),
                    makeMethodName(value_expr), ⟨⟩),
            _ →
                TM_MethodInvocation(
                    TM_NoOptionalExpression,
                    makeMethodName(value_expr),
                    makeArgumentList(vel))
        end,
    TM_ParenthesizedExpr(value_expr) →
        TM_ParenthesizedExpression(
            makeExpression(value_expr)),
    Make_TM_ListExpr(ele) →
        case ele of
            TM_EnumeratedListExpr(vel) →
                if vel = ⟨⟩
                then
                    TM_ClassInstanceCreation(
                        TM_NoOptionalExpression,
                        matchTypeReferenceType(
                            getTypeEvaluatorForTypeEvaluation(
                                ele), true, false), ⟨⟩)
                else
                    TM_ClassInstanceCreation(
                        TM_NoOptionalExpression,
                        matchTypeReferenceType(
                            getTypeEvaluatorForTypeEvaluation(
                                ele), true, false),
                        ⟨TM_ArrayCreation(
                            matchTypeInner(
                                getTypeEvaluatorForTypeEvaluation(
                                    ele), true, false),
                            TM_NoOptionalExpression,
                            makeArgumentList(vel))⟩)
                end,
```

```
            _ →
                Make_TM_JavaName(
                    Make_TM_SimpleName(
                        mk_TM_SimpleName("UNKNOWN_LIST_EXPR")))
        end,
    Make_TM_ValueOrVariableName(vovn) →
        case getTypeEvaluatorForTypeEvaluation(vovn) of
            TM_TypeEvaluator_TM_Constructor(c) →
                TM_ClassInstanceCreation(
                    TM_NoOptionalExpression,
                    mk_TM_ReferenceType(
                        Make_TM_SimpleName(
                            rslIdToSimpleName(id(c))),
                        TM_NoOptionalReferenceType), ⟨⟩),
            /*TM_TypeEvaluator_TM_TypeExpr(te) → Make_TM_
            JavaName(Make_TM_SimpleName(mk_TM_SimpleName(
            getText(id(vovn))))),*/
            _ →
                Make_TM_JavaName(
                    Make_TM_SimpleName(
                        mk_TM_SimpleName(getText(id(vovn)))))
            /*_ → Make_TM_JavaValueLiteral(Make_TM_JavaValueLiteralInteA
            ger(mk_TM_JavaValueLiteralInteger("UNKNOWN_VALUE_
            OR_VARIABLE_NAME")))*/
            end,
    _ →
        Make_TM_JavaValueLiteral(
            Make_TM_JavaValueLiteralInteger(
                mk_TM_JavaValueLiteralInteger(
                    "UNKNOWN_EXPRESSION")))
end,

makeMethodName : TM_ValueExpr → TM_SimpleName
makeMethodName(ve) ≡
    case ve of
        Make_TM_ValueOrVariableName(vovn) →
            mk_TM_SimpleName(getText(id(vovn))),
        _ → mk_TM_SimpleName("UNKNOWN_METHOD_NAME")
    end,

makeArgumentList :
```

```
    TM_ValueExpr* → TM_Expression*
makeArgumentList(vel) ≡
    if vel = ⟨⟩ then ⟨⟩
    else
        ⟨makeExpression(hd vel)⟩ ⌢
        makeArgumentList(tl vel)
    end,

makeMainMethodDeclaration :
    TM_Decl* → TM_MethodDeclaration*
makeMainMethodDeclaration(dl) ≡
    if makeMainMethodDeclaration1(dl) = ⟨⟩ then ⟨⟩
    else
        ⟨mk_TM_MethodDeclaration(
                ⟨TM_JAVA_PUBLIC, TM_JAVA_STATIC⟩,
                mk_TM_SimpleName(″main″),
                Make_TM_PrimitiveType(TM_JAVA_VOID),
                ⟨mk_TM_SingleVariableDeclaration(
                        ⟨⟩,
                        Make_TM_ArrayType(
                            mk_TM_ArrayType(
                                Make_TM_ReferenceType(
                                    mk_TM_ReferenceType(
                                        Make_TM_SimpleName(
                                            mk_TM_SimpleName(″String″)
                                        ), TM_NoOptionalReferenceType
                                    ))))), mk_TM_SimpleName(″args″),
                        TM_NoOptionalExpression)⟩,
                Make_TM_OptionalBlock(
                      mk_TM_Block(makeMainMethodDeclaration1(dl))
                )⟩
    end,

makeMainMethodDeclaration1 :
    TM_Decl* → TM_Statement*
makeMainMethodDeclaration1(dl) ≡
    if dl = ⟨⟩ then ⟨⟩
    else
        case hd dl of
            TM_TestDecl(tdl) →
                makeMainMethodDeclaration2(tdl) ⌢
```

```
            makeMainMethodDeclaration1(tl dl),
        _ → makeMainMethodDeclaration1(tl dl)
    end
  end,

makeMainMethodDeclaration2 :
  TM_TestDef* → TM_Statement*
makeMainMethodDeclaration2(tdl) ≡
  if tdl = ⟨⟩ then ⟨⟩
  else
    case hd tdl of
      TM_TestCase(id, value_expr) →
        makeTestDeclaration(id, value_expr) ⌢
        makeMainMethodDeclaration2(tl tdl),
      _ → makeMainMethodDeclaration2(tl tdl)
    end
  end,

makeTestDeclaration :
  TM_Id × TM_ValueExpr → TM_Statement*
makeTestDeclaration(id, ve) ≡
  ⟨TM_ExpressionStatement(
      TM_MethodInvocation(
        Make_TM_OptionalExpression(
          Make_TM_JavaName(
            Make_TM_QualifiedName(
              mk_TM_QualifiedName(
                Make_TM_SimpleName(
                  mk_TM_SimpleName("System")),
                mk_TM_SimpleName("out"))))),
        mk_TM_SimpleName("println"),
        ⟨TM_InfixExpression(
          Make_TM_JavaValueLiteral(
            Make_TM_JavaValueLiteralString(
              mk_TM_JavaValueLiteralString(
                "[" ⌢ getText(id) ⌢ "]: "))),
          TM_JAVA_PLUS, makeExpression(ve))⟩)))⟩,

makeCompilationUnitList :
  TM_ClassExpr × TM_OptionalPackageDeclaration ×
  TM_ImportDeclaration* ×
```

```
    TM_OptionalSimpleName × TM_OptionalId →
        TM_CompilationUnit*
makeCompilationUnitList(ce, opd, idl, ext, visitor) ≡
    case ce of
        TM_BasicClassExpr(dl) →
            makeCompilationUnitList1(
                dl, opd, idl, ext, visitor),
        _ → ⟨⟩
    end,

makeCompilationUnitList1 :
    TM_Decl* × TM_OptionalPackageDeclaration ×
    TM_ImportDeclaration* ×
    TM_OptionalSimpleName × TM_OptionalId →
        TM_CompilationUnit*
makeCompilationUnitList1(dl, opd, idl, ext, visitor) ≡
    if dl = ⟨⟩ then ⟨⟩
    else
        case hd dl of
            TM_TypeDecl(tdl) →
                makeCompilationUnitList2(
                    tdl, opd, idl, ext, visitor) ⌢
                makeCompilationUnitList1(
                    tl dl, opd, idl, ext, visitor),
            _ →
                makeCompilationUnitList1(
                    tl dl, opd, idl, ext, visitor)
        end
    end,

makeCompilationUnitList2 :
    TM_TypeDef* × TM_OptionalPackageDeclaration ×
    TM_ImportDeclaration* ×
    TM_OptionalSimpleName × TM_OptionalId →
        TM_CompilationUnit*
makeCompilationUnitList2(tdl, opd, idl, ext, visitor) ≡
    if tdl = ⟨⟩ then ⟨⟩
    else
        makeClassDeclaration1(
            hd tdl, opd, idl, ext, visitor) ⌢
        makeCompilationUnitList2(
```

```
            tl tdl, opd, idl, ext, visitor)
    end,

makeClassDeclaration1 :
    TM_TypeDef × TM_OptionalPackageDeclaration ×
    TM_ImportDeclaration* ×
    TM_OptionalSimpleName × TM_OptionalId →
        TM_CompilationUnit*
makeClassDeclaration1(td, opd, idl, ext, visitor) ≡
    case td of
        TM_VariantDef(id, vl) →
            ⟨makeClassDeclaration2(
                    id, opd, idl, ext, visitor)⟩ ⌢
            makeClassDeclarationList(
                vl, id, opd, idl,
                Make_TM_OptionalSimpleName(
                    rslIdToSimpleName(id)), visitor),
        TM_ShortRecordDef(id, cks) →
            case visitor of
                Make_TM_Id(visitorId) →
                    ⟨mk_TM_CompilationUnit(
                            opd, idl,
                            ⟨TM_ClassDeclaration(
                                    ⟨TM_JAVA_PUBLIC⟩,
                                    rslIdToSimpleName(id), ext, ⟨⟩,
                                    makeConstructorDeclaration(id, cks),
                                    makeStandardMethodDeclarationList(
                                      id, cks) ⌢
                                    makeVisitorMethod(
                                      id, visitorId, false),
                                    makeFieldDeclaration(cks, 0), ⟨⟩)
                            ⟩)⟩,
                TM_NoOptionalId →
                    ⟨mk_TM_CompilationUnit(
                            opd, idl,
                            ⟨TM_ClassDeclaration(
                                    ⟨TM_JAVA_PUBLIC⟩,
                                    rslIdToSimpleName(id), ext, ⟨⟩,
                                    makeConstructorDeclaration(id, cks),
                                    makeStandardMethodDeclarationList(
                                      id, cks),
```

```
                        makeFieldDeclaration(cks, 0), ⟨⟩)
                ⟩)⟩
    end,
TM_SortDef(id) →
    case visitor of
        Make_TM_Id(visitorId) →
            ⟨mk_TM_CompilationUnit(
                opd, idl,
                ⟨TM_ClassDeclaration(
                    ⟨TM_JAVA_PUBLIC⟩,
                    rslIdToSimpleName(id), ext, ⟨⟩,
                    ⟨⟩,
                    ⟨mk_TM_MethodDeclaration(
                        ⟨TM_JAVA_PUBLIC⟩,
                        mk_TM_SimpleName("equals"),
                        Make_TM_PrimitiveType(
                          TM_JAVA_BOOL),
                        ⟨mk_TM_SingleVariableDeclaration(
                            ⟨⟩,
                            Make_TM_ReferenceType(
                              mk_TM_ReferenceType
                              (
                    Make_TM_SimpleName(
                        mk_TM_SimpleName(
                          "Object")),
                    TM_NoOptionalReferenceType)),
                                mk_TM_SimpleName("o"),
                                TM_NoOptionalExpression)
                        ⟩,
                        Make_TM_OptionalBlock(
                          mk_TM_Block(
                            ⟨TM_ReturnStatement(
                                TM_InstanceOfExpression(
                                  Make_TM_JavaName(
                                    Make_TM_SimpleName(
                                      mk_TM_SimpleName(
                                        "o"))),
                                  Make_TM_ReferenceType(
                                    mk_TM_ReferenceType
                                    (
                                Make_TM_SimpleName(
```

```
                    mk_TM_SimpleName(
                      getText
                      (
                    id))),
                  TM_NoOptionalReferenceType)))
                      )⟩⟩⟩⟩⟩),
              mk_TM_MethodDeclaration(
                ⟨TM_JAVA_PUBLIC⟩,
                mk_TM_SimpleName("toString"),
                Make_TM_ReferenceType(
                  mk_TM_ReferenceType
                  (
Make_TM_SimpleName(
    mk_TM_SimpleName(
        "String")),
TM_NoOptionalReferenceType)), ⟨⟩,
                Make_TM_OptionalBlock(
                  mk_TM_Block(
                    ⟨TM_ReturnStatement(
                        Make_TM_JavaValueLiteral(
                          Make_TM_JavaValueLiteralString(
                            mk_TM_JavaValueLiteralString(
                              getText(
                                id)))))
                    ⟩⟩⟩⟩),
              mk_TM_MethodDeclaration(
                ⟨TM_JAVA_PUBLIC⟩,
                mk_TM_SimpleName("accept"),
                Make_TM_PrimitiveType(
                  TM_JAVA_BOOL),
                ⟨mk_TM_SingleVariableDeclaration(
                    ⟨⟩,
                    Make_TM_ReferenceType(
                      mk_TM_ReferenceType
                      (
        Make_TM_SimpleName(
            rslIdToSimpleName(
              visitorId)),
        TM_NoOptionalReferenceType)),
                    mk_TM_SimpleName(
                      "visitor"),
```

```
                                    TM_NoOptionalExpression)
                                ⟩,
                                Make_TM_OptionalBlock(
                                  mk_TM_Block(
                                    ⟨TM_ExpressionStatement(
                                        TM_MethodInvocation(
                                          Make_TM_OptionalExpression(
                                            Make_TM_JavaName(
                                              Make_TM_SimpleName(
                                                mk_TM_SimpleName
                                                (
                                          "visitor")))),
                                            mk_TM_SimpleName(
                                              "visit" ^
                                              getText(id)),
                                            ⟨TM_ThisExpression
                                              (
                                        TM_NoOptionalSimpleName)⟩)))
                                      ⟩)))⟩, ⟨⟩, ⟨⟩)⟩)
    ⟩,
  TM_NoOptionalId →
    ⟨mk_TM_CompilationUnit(
          opd, idl,
          ⟨TM_ClassDeclaration(
                ⟨TM_JAVA_PUBLIC⟩,
                rslIdToSimpleName(id), ext, ⟨⟩,
                ⟨⟩,
                ⟨mk_TM_MethodDeclaration(
                    ⟨TM_JAVA_PUBLIC⟩,
                    mk_TM_SimpleName("equals"),
                    Make_TM_PrimitiveType(
                      TM_JAVA_BOOL),
                    ⟨mk_TM_SingleVariableDeclaration(
                        ⟨⟩,
                        Make_TM_ReferenceType(
                          mk_TM_ReferenceType
                          (
                  Make_TM_SimpleName(
                      mk_TM_SimpleName(
                        "Object")),
                  TM_NoOptionalReferenceType)),
```

```
                                    mk_TM_SimpleName("o"),
                                    TM_NoOptionalExpression)
                                ⟩,
                                Make_TM_OptionalBlock(
                                  mk_TM_Block(
                                    ⟨TM_ReturnStatement(
                                        TM_InstanceOfExpression(
                                          Make_TM_JavaName(
                                            Make_TM_SimpleName(
                                              mk_TM_SimpleName(
                                                "o"))),
                                          Make_TM_ReferenceType(
                                            mk_TM_ReferenceType
                                            (
                                     Make_TM_SimpleName(
                                       mk_TM_SimpleName(
                                         getText
                                         (
                                       id))),
                                     TM_NoOptionalReferenceType)))
                                           )⟩⟩⟩⟩⟩),
                              mk_TM_MethodDeclaration(
                                ⟨TM_JAVA_PUBLIC⟩,
                                mk_TM_SimpleName("toString"),
                                Make_TM_ReferenceType(
                                  mk_TM_ReferenceType
                                  (
                 Make_TM_SimpleName(
                     mk_TM_SimpleName(
                         "String")),
                 TM_NoOptionalReferenceType)), ⟨⟩,
                                Make_TM_OptionalBlock(
                                  mk_TM_Block(
                                    ⟨TM_ReturnStatement(
                                        Make_TM_JavaValueLiteral(
                                          Make_TM_JavaValueLiteralString(
                                            mk_TM_JavaValueLiteralString(
                                              getText(
                                                id)))))
                                    ⟩⟩⟩⟩⟩, ⟨⟩, ⟨⟩⟩⟩)
⟩
```

```
        end
    end,

makeClassDeclaration2 :
    TM_Id × TM_OptionalPackageDeclaration ×
    TM_ImportDeclaration* ×
    TM_OptionalSimpleName × TM_OptionalId →
        TM_CompilationUnit
makeClassDeclaration2(id, opd, idl, ext, visitor) ≡
    case visitor of
        Make_TM_Id(visitorId) →
            mk_TM_CompilationUnit(
                opd, idl,
                ⟨TM_ClassDeclaration(
                        ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                        rslIdToSimpleName(id), ext, ⟨⟩, ⟨⟩,
                        ⟨mk_TM_MethodDeclaration(
                                ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                                mk_TM_SimpleName(″equals″),
                                Make_TM_PrimitiveType(TM_JAVA_BOOL),
                                ⟨mk_TM_SingleVariableDeclaration(
                                    ⟨⟩,
                                    Make_TM_ReferenceType(
                                      mk_TM_ReferenceType
                                      (
                    Make_TM_SimpleName(
                          mk_TM_SimpleName(
                              ″Object″)),
                    TM_NoOptionalReferenceType)),
                                        mk_TM_SimpleName(″o″),
                                        TM_NoOptionalExpression)⟩,
                                TM_NoOptionalBlock),
                            mk_TM_MethodDeclaration(
                                ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                                mk_TM_SimpleName(″toString″),
                                Make_TM_ReferenceType(
                                  mk_TM_ReferenceType(
                                    Make_TM_SimpleName(
                                      mk_TM_SimpleName(″String″)),
                                    TM_NoOptionalReferenceType)),
                                ⟨⟩, TM_NoOptionalBlock)⟩ ⌢
```

```
                    makeVisitorMethod(id, visitorId, true),
                    ⟨⟩, ⟨⟩)⟩)),
          TM_NoOptionalId →
            mk_TM_CompilationUnit(
              opd, idl,
              ⟨TM_ClassDeclaration(
                    ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                    rslIdToSimpleName(id), ext, ⟨⟩, ⟨⟩,
                    ⟨mk_TM_MethodDeclaration(
                          ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                          mk_TM_SimpleName("equals"),
                          Make_TM_PrimitiveType(TM_JAVA_BOOL),
                          ⟨mk_TM_SingleVariableDeclaration(
                              ⟨⟩,
                              Make_TM_ReferenceType(
                                mk_TM_ReferenceType
                                (
                    Make_TM_SimpleName(
                          mk_TM_SimpleName(
                            "Object")),
                    TM_NoOptionalReferenceType)),
                                mk_TM_SimpleName("o"),
                                TM_NoOptionalExpression)⟩,
                          TM_NoOptionalBlock),
                        mk_TM_MethodDeclaration(
                          ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                          mk_TM_SimpleName("toString"),
                          Make_TM_ReferenceType(
                            mk_TM_ReferenceType(
                              Make_TM_SimpleName(
                                mk_TM_SimpleName("String")),
                              TM_NoOptionalReferenceType)),
                          ⟨⟩, TM_NoOptionalBlock)⟩, ⟨⟩,
                    ⟨⟩)⟩)
    end,

  makeClassDeclarationList :
    TM_Variant* × TM_Id ×
    TM_OptionalPackageDeclaration ×
    TM_ImportDeclaration* ×
    TM_OptionalSimpleName × TM_OptionalId →
```

```
        TM_CompilationUnit*
    makeClassDeclarationList(
        vl, id, opd, idl, ext, visitor) ≡
      if vl = ⟨⟩ then ⟨⟩
      else
        case hd vl of
          Make_TM_Constructor(c) →
            case visitor of
              Make_TM_Id(visitorId) →
                ⟨mk_TM_CompilationUnit(
                      opd, idl,
                      ⟨TM_ClassDeclaration(
                            ⟨TM_JAVA_PUBLIC⟩,
                            rslIdToSimpleName(id(c)), ext, ⟨
                            ⟩, ⟨⟩,
                            makeStandardMethodDeclarationListConstant(
                              id(c)) ⌢
                            makeVisitorMethod(
                              id(c), visitorId, false),
                            ⟨⟩, ⟨⟩)⟩)⟩ ⌢
                  makeClassDeclarationList(
                      tl vl, id, opd, idl, ext, visitor),
              TM_NoOptionalId →
                ⟨mk_TM_CompilationUnit(
                      opd, idl,
                      ⟨TM_ClassDeclaration(
                            ⟨TM_JAVA_PUBLIC⟩,
                            rslIdToSimpleName(id(c)), ext, ⟨
                            ⟩, ⟨⟩,
                            makeStandardMethodDeclarationListConstant(
                              id(c)), ⟨⟩, ⟨⟩)⟩)⟩ ⌢
                  makeClassDeclarationList(
                      tl vl, id, opd, idl, ext, visitor)
              end,
          TM_RecordVariant(c, ckl) →
            case visitor of
              Make_TM_Id(visitorId) →
                ⟨mk_TM_CompilationUnit(
                      opd, idl,
                      ⟨TM_ClassDeclaration(
                            ⟨TM_JAVA_PUBLIC⟩,
```

```
                                    rslIdToSimpleName(id(c)), ext, ⟨
                                    ⟩,
                                    makeConstructorDeclaration(
                                      id(c), ckl),
                                    makeStandardMethodDeclarationList(
                                      id(c), ckl) ⌢
                                    makeVisitorMethod(
                                      id(c), visitorId, false),
                                    makeFieldDeclaration(ckl, 0), ⟨
                                    ⟩⟩⟩⟩⟩ ⌢
                        makeClassDeclarationList(
                            tl vl, id, opd, idl, ext, visitor),
                    TM_NoOptionalId →
                        ⟨mk_TM_CompilationUnit(
                                opd, idl,
                                ⟨TM_ClassDeclaration(
                                    ⟨TM_JAVA_PUBLIC⟩,
                                    rslIdToSimpleName(id(c)), ext, ⟨
                                    ⟩,
                                    makeConstructorDeclaration(
                                      id(c), ckl),
                                    makeStandardMethodDeclarationList(
                                      id(c), ckl),
                                    makeFieldDeclaration(ckl, 0), ⟨
                                    ⟩⟩⟩⟩⟩ ⌢
                        makeClassDeclarationList(
                            tl vl, id, opd, idl, ext, visitor)
                end
            end
        end,

makeStandardMethodDeclarationListConstant :
    TM_Id → TM_MethodDeclaration*
makeStandardMethodDeclarationListConstant(id) ≡
    ⟨makeEqualsMethodConstant(id)⟩ ⌢
    ⟨makeToStringMethodConstant(id)⟩,

makeEqualsMethodConstant :
    TM_Id → TM_MethodDeclaration
makeEqualsMethodConstant(id) ≡
    mk_TM_MethodDeclaration(
```

```
        ⟨TM_JAVA_PUBLIC⟩, mk_TM_SimpleName(″equals″),
        Make_TM_PrimitiveType(TM_JAVA_BOOL),
        ⟨mk_TM_SingleVariableDeclaration(
                ⟨⟩,
                Make_TM_ReferenceType(
                    mk_TM_ReferenceType(
                        Make_TM_SimpleName(
                            mk_TM_SimpleName(″Object″)),
                        TM_NoOptionalReferenceType)),
                mk_TM_SimpleName(″o″),
                TM_NoOptionalExpression)⟩,
        Make_TM_OptionalBlock(
            mk_TM_Block(
                ⟨TM_ReturnStatement(
                        TM_InstanceOfExpression(
                            Make_TM_JavaName(
                              Make_TM_SimpleName(
                                mk_TM_SimpleName(″o″))),
                            Make_TM_ReferenceType(
                              mk_TM_ReferenceType(
                                Make_TM_SimpleName(
                                  mk_TM_SimpleName(
                                    getText(id))),
                                TM_NoOptionalReferenceType))))
                ⟩))),

makeToStringMethodConstant :
    TM_Id → TM_MethodDeclaration
makeToStringMethodConstant(id) ≡
    mk_TM_MethodDeclaration(
        ⟨TM_JAVA_PUBLIC⟩, mk_TM_SimpleName(″toString″),
        Make_TM_ReferenceType(
            mk_TM_ReferenceType(
                Make_TM_SimpleName(
                    mk_TM_SimpleName(″String″)),
                TM_NoOptionalReferenceType)), ⟨⟩,
        Make_TM_OptionalBlock(
            mk_TM_Block(
                ⟨TM_ReturnStatement(
                        Make_TM_JavaValueLiteral(
                            Make_TM_JavaValueLiteralString(
```

```
                                    mk_TM_JavaValueLiteralString(
                                      getText(id)))))))))),

makeFieldDeclaration :
   TM_ComponentKind* × Int →
      TM_FieldDeclaration*
makeFieldDeclaration(ckl, i) ≡
   if ckl = ⟨⟩ then ⟨⟩
   else
      ⟨mk_TM_FieldDeclaration(
            ⟨TM_JAVA_PRIVATE⟩,
            matchType(type_expr(hd ckl), false, true),
            mk_TM_VariableDeclarationFragment(
                  mk_TM_SimpleName("_v" ⌢ convert(i)),
                  TM_NoOptionalExpression))⟩ ⌢
      makeFieldDeclaration(tl ckl, i + 1)
   end,

makeConstructorDeclaration :
   TM_Id × TM_ComponentKind* →
      TM_ConstructorDeclaration*
makeConstructorDeclaration(id, ckl) ≡
   ⟨mk_TM_ConstructorDeclaration(
         ⟨TM_JAVA_PUBLIC⟩, rslIdToSimpleName(id),
         makeArgumentList(ckl, 0),
         mk_TM_Block(
               makeConstructorStatementList(ckl, 0)))⟩,

makeConstructorStatementList :
   TM_ComponentKind* × Int → TM_Statement*
makeConstructorStatementList(ckl, i) ≡
   if ckl = ⟨⟩ then ⟨⟩
   else
      ⟨TM_ExpressionStatement(
            TM_AssignmentExpression(
                  TM_FieldAccessExpression(
                        Make_TM_OptionalExpression(
                              TM_ThisExpression(
                                    TM_NoOptionalSimpleName)),
                        mk_TM_SimpleName("_v" ⌢ convert(i))),
                  TM_JAVA_ASSIGNMENT_OP_EQUAL,
```

```
                    Make_TM_JavaName(
                        Make_TM_SimpleName(
                            mk_TM_SimpleName("_v" ⌢ convert(i))))
                ))⟩ ⌢
        makeConstructorStatementList(tl ckl, i + 1)
    end,

makeStandardMethodDeclarationList :
    TM_Id × TM_ComponentKind* →
        TM_MethodDeclaration*
makeStandardMethodDeclarationList(id, ckl) ≡
    ⟨mk_TM_MethodDeclaration(
            ⟨TM_JAVA_PUBLIC⟩, mk_TM_SimpleName("equals"),
            Make_TM_PrimitiveType(TM_JAVA_BOOL),
            ⟨mk_TM_SingleVariableDeclaration(
                    ⟨⟩,
                    Make_TM_ReferenceType(
                        mk_TM_ReferenceType(
                            Make_TM_SimpleName(
                              mk_TM_SimpleName("Object")),
                            TM_NoOptionalReferenceType)),
                    mk_TM_SimpleName("o"),
                    TM_NoOptionalExpression)⟩,
            Make_TM_OptionalBlock(
                mk_TM_Block(
                    makeStatementListEqualsMethod(id, ckl) ⌢
                    ⟨TM_ReturnStatement(
                            Make_TM_JavaValueLiteral(
                              Make_TM_JavaValueLiteralBool(
                                mk_TM_JavaValueLiteralBool(
                                  "true")))))⟩))),
        mk_TM_MethodDeclaration(
            ⟨TM_JAVA_PUBLIC⟩,
            mk_TM_SimpleName("toString"),
            Make_TM_ReferenceType(
                mk_TM_ReferenceType(
                    Make_TM_SimpleName(
                        mk_TM_SimpleName("String")),
                    TM_NoOptionalReferenceType)), ⟨⟩,
            Make_TM_OptionalBlock(
                mk_TM_Block(
```

```
                    ⟨TM_ReturnStatement(
                            TM_InfixExpression(
                              Make_TM_JavaValueLiteral(
                                Make_TM_JavaValueLiteralString(
                                  mk_TM_JavaValueLiteralString(
                                    getText(id) ⌢ ″(″))),
                              TM_JAVA_PLUS,
                              makeExpressionToStringMethod(ckl))
                        ))))⟩ ⌢
    makeAccessorMethodDeclarationList(ckl, 0),

makeStatementListEqualsMethod :
    TM_Id × TM_ComponentKind* → TM_Statement*
makeStatementListEqualsMethod(id, ckl) ≡
    if ckl = ⟨⟩ then ⟨⟩
    else
      case matchType(type_expr(hd ckl), false, false) of
        Make_TM_PrimitiveType(t) →
          ⟨TM_IfStatement(
                TM_PrefixExpression(
                    TM_JAVA_NOT,
                    TM_ParenthesizedExpression(
                      TM_InfixExpression(
                        TM_MethodInvocation(
                          Make_TM_OptionalExpression(
                            TM_ThisExpression(
                              TM_NoOptionalSimpleName)
                          ),
                          makeSimpleNameFromDestructor(
                            optional_destructor(hd ckl)
                          ), ⟨⟩), TM_JAVA_EQUALS,
                        TM_MethodInvocation(
                          Make_TM_OptionalExpression(
                            TM_ParenthesizedExpression(
                              TM_CastExpression(
                                Make_TM_ReferenceType(
                                  mk_TM_ReferenceType
                                  (
                        Make_TM_SimpleName(
                          rslIdToSimpleName(
                            id)),
```

```
                                TM_NoOptionalReferenceType)),
                                    Make_TM_JavaName(
                                      Make_TM_SimpleName(
                                        mk_TM_SimpleName(
                                          "o"))))),
                                makeSimpleNameFromDestructor(
                                  optional_destructor(hd ckl)
                                ), ⟨⟩)))),
                        mk_TM_Block(
                            ⟨TM_ReturnStatement(
                                Make_TM_JavaValueLiteral(
                                  Make_TM_JavaValueLiteralBool(
                                    mk_TM_JavaValueLiteralBool(
                                      "false"))))),
                            TM_NoOptionalBlock)⟩ ⌢
                    makeStatementListEqualsMethod(id, tl ckl),
                Make_TM_ReferenceType(t) →
                    ⟨TM_IfStatement(
                            TM_MethodInvocation(
                                Make_TM_OptionalExpression(
                                    TM_MethodInvocation(
                                      Make_TM_OptionalExpression(
                                        TM_ThisExpression(
                                          TM_NoOptionalSimpleName)),
                                      makeSimpleNameFromDestructor(
                                        optional_destructor(hd ckl)),
                                      ⟨⟩)),
                                mk_TM_SimpleName("equals"),
                                ⟨TM_MethodInvocation(
                                    Make_TM_OptionalExpression(
                                      TM_ParenthesizedExpression(
                                        TM_CastExpression(
                                          Make_TM_ReferenceType(
                                            mk_TM_ReferenceType
                                            (
                                  Make_TM_SimpleName(
                                    rslIdToSimpleName(
                                      id)),
                                  TM_NoOptionalReferenceType)),
                                            Make_TM_JavaName(
                                              Make_TM_SimpleName(
```

```
                                                mk_TM_SimpleName(
                                                  "o"))))))),
                                        makeSimpleNameFromDestructor(
                                          optional_destructor(hd ckl)),
                                        ⟨⟩)))),
                              mk_TM_Block(
                                    ⟨TM_ReturnStatement(
                                          Make_TM_JavaValueLiteral(
                                            Make_TM_JavaValueLiteralBool(
                                              mk_TM_JavaValueLiteralBool(
                                                "false"))))))⟩),
                              TM_NoOptionalBlock)⟩ ⌢
                        makeStatementListEqualsMethod(id, tl ckl)
          end
     end,

makeVisitorMethod :
     TM_Id × TM_Id × Bool → TM_MethodDeclaration*
makeVisitorMethod(id, visitor, abstractMethod) ≡
     if abstractMethod
     then
          ⟨mk_TM_MethodDeclaration(
                ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                mk_TM_SimpleName("accept"),
                Make_TM_PrimitiveType(TM_JAVA_VOID),
                ⟨mk_TM_SingleVariableDeclaration(
                      ⟨⟩,
                      Make_TM_ReferenceType(
                          mk_TM_ReferenceType(
                              Make_TM_SimpleName(
                                rslIdToSimpleName(visitor)),
                              TM_NoOptionalReferenceType)),
                      mk_TM_SimpleName("visitor"),
                      TM_NoOptionalExpression)⟩,
                TM_NoOptionalBlock)⟩
     else
          ⟨mk_TM_MethodDeclaration(
                ⟨TM_JAVA_PUBLIC⟩,
                mk_TM_SimpleName("accept"),
                Make_TM_PrimitiveType(TM_JAVA_VOID),
                ⟨mk_TM_SingleVariableDeclaration(
```

```
                    ⟨⟩,
                    Make_TM_ReferenceType(
                        mk_TM_ReferenceType(
                            Make_TM_SimpleName(
                              rslIdToSimpleName(visitor)),
                            TM_NoOptionalReferenceType)),
                    mk_TM_SimpleName(″visitor″),
                    TM_NoOptionalExpression)⟩,
                Make_TM_OptionalBlock(
                    mk_TM_Block(
                        ⟨TM_ExpressionStatement(
                            TM_MethodInvocation(
                              Make_TM_OptionalExpression(
                                Make_TM_JavaName(
                                  Make_TM_SimpleName(
                                    mk_TM_SimpleName(
                                      ″visitor″)))),
                              mk_TM_SimpleName(
                                ″visit″ ⌢ getText(id)),
                              ⟨TM_ThisExpression(
                                  TM_NoOptionalSimpleName)⟩)
                        )⟩⟩⟩⟩⟩⟩
    end,

makeSimpleNameFromDestructor :
    TM_OptionalDestructor → TM_SimpleName
makeSimpleNameFromDestructor(od) ≡
    case od of
        TM_Destructor(d) → rslIdToSimpleName(d),
        TM_NoDestructor →
            mk_TM_SimpleName(″UNKNOWN_METHOD_NAME″)
    end,

makeExpressionToStringMethod :
    TM_ComponentKind* → TM_Expression
makeExpressionToStringMethod(ckl) ≡
    if ckl = ⟨⟩
    then
        Make_TM_JavaValueLiteral(
            Make_TM_JavaValueLiteralString(
                mk_TM_JavaValueLiteralString(″)″)))
```

```
else
  case matchType(type_expr(hd ckl), false, false) of
    Make_TM_PrimitiveType(pt) →
      if tl ckl = ⟨⟩
      then
        TM_InfixExpression(
          TM_MethodInvocation(
            Make_TM_OptionalExpression(
              TM_ThisExpression(
                TM_NoOptionalSimpleName)),
            makeSimpleNameFromDestructor(
              optional_destructor(hd ckl)), ⟨⟩),
          TM_JAVA_PLUS,
          makeExpressionToStringMethod(tl ckl))
      else
        TM_InfixExpression(
          TM_MethodInvocation(
            Make_TM_OptionalExpression(
              TM_ThisExpression(
                TM_NoOptionalSimpleName)),
            makeSimpleNameFromDestructor(
              optional_destructor(hd ckl)), ⟨⟩),
          TM_JAVA_PLUS,
          TM_InfixExpression(
            Make_TM_JavaValueLiteral(
              Make_TM_JavaValueLiteralString(
                mk_TM_JavaValueLiteralString(
                  ", ")), TM_JAVA_PLUS,
            makeExpressionToStringMethod(tl ckl)))
      end,
    Make_TM_ReferenceType(rt) →
      if tl ckl = ⟨⟩
      then
        TM_InfixExpression(
          TM_MethodInvocation(
            Make_TM_OptionalExpression(
              TM_MethodInvocation(
                Make_TM_OptionalExpression(
                  TM_ThisExpression(
                    TM_NoOptionalSimpleName)),
                makeSimpleNameFromDestructor(
```

```
                                    optional_destructor(hd ckl)),
                                ⟨⟩)),
                            mk_TM_SimpleName(″toString″), ⟨⟩),
                        TM_JAVA_PLUS,
                        makeExpressionToStringMethod(tl ckl))
                else
                    TM_InfixExpression(
                        TM_MethodInvocation(
                            Make_TM_OptionalExpression(
                                TM_MethodInvocation(
                                    Make_TM_OptionalExpression(
                                        TM_ThisExpression(
                                            TM_NoOptionalSimpleName)),
                                    makeSimpleNameFromDestructor(
                                        optional_destructor(hd ckl)),
                                    ⟨⟩)),
                            mk_TM_SimpleName(″toString″), ⟨⟩),
                        TM_JAVA_PLUS,
                        TM_InfixExpression(
                            Make_TM_JavaValueLiteral(
                                Make_TM_JavaValueLiteralString(
                                    mk_TM_JavaValueLiteralString(
                                        ″, ″))), TM_JAVA_PLUS,
                            makeExpressionToStringMethod(tl ckl)))
                end
            end
        end,

    makeAccessorMethodDeclarationList :
        TM_ComponentKind* × Int →
            TM_MethodDeclaration*
    makeAccessorMethodDeclarationList(ckl, i) ≡
        if ckl = ⟨⟩ then ⟨⟩
        else
            ⟨mk_TM_MethodDeclaration(
                    ⟨TM_JAVA_PUBLIC⟩,
                    makeSimpleNameFromDestructor(
                        optional_destructor(hd ckl)),
                    matchType(type_expr(hd ckl), false, true), ⟨
                    ⟩,
                    Make_TM_OptionalBlock(
```

```
                    mk_TM_Block(
                        ⟨TM_ReturnStatement(
                                Make_TM_JavaName(
                                  Make_TM_SimpleName(
                                    mk_TM_SimpleName(
                                      "_v" ⌢ convert(i)))))⟩))))
        ⟩ ⌢
        makeAccessorMethodDeclarationList(tl ckl, i + 1)
    end,

matchTypeInner :
    TM_TypeEvaluator × Bool × Bool → TM_JavaType
matchTypeInner(te, useWrapper, useInterface) ≡
    case te of
        TM_TypeEvaluator_TM_TypeExpr(inner_type_expr) →
            case inner_type_expr of
                TM_TypeExprList(tel) →
                    case tel of
                        TM_FiniteListTypeExpr(flte) →
                            matchType(flte, useWrapper, useInterface),
                        _ →
                            Make_TM_ReferenceType(
                                mk_TM_ReferenceType(
                                    Make_TM_SimpleName(
                                      mk_TM_SimpleName
                                      (
                    "UNKNOWN_TYPE_MATCH_TYPE_INNER_1")),
                                                TM_NoOptionalReferenceType))
                    end,
                _ →
                    Make_TM_ReferenceType(
                        mk_TM_ReferenceType(
                            Make_TM_SimpleName(
                                mk_TM_SimpleName(
                                  "UNKNOWN_TYPE_MATCH_TYPE_INNER_2"
                                )), TM_NoOptionalReferenceType))
            end,
        _ →
            Make_TM_ReferenceType(
                mk_TM_ReferenceType(
                    Make_TM_SimpleName(
```

```
                    mk_TM_SimpleName(
                        "UNKNOWN_TYPE_MATCH_TYPE_INNER_3")),
                TM_NoOptionalReferenceType))
  end,

matchType :
  TM_TypeEvaluator × Bool × Bool → TM_JavaType
matchType(te, useWrapper, useInterface) ≡
  case te of
    TM_TypeEvaluator_TM_TypeExpr(type_expr) →
      matchType(type_expr, useWrapper, useInterface),
    TM_TypeEvaluator_TM_Constructor(c) →
      Make_TM_ReferenceType(
        mk_TM_ReferenceType(
          Make_TM_SimpleName(
            rslIdToSimpleName(id(c))),
          TM_NoOptionalReferenceType))
  end,

matchType : TM_TypeExpr × Bool × Bool → TM_JavaType
matchType(te, useWrapper, useInterface) ≡
  case te of
    TM_TypeLiteral(literal) →
      case literal of
        TM_RSL_UNIT →
          Make_TM_PrimitiveType(TM_JAVA_VOID),
        TM_RSL_INT →
          if useWrapper
          then
            Make_TM_ReferenceType(
              mk_TM_ReferenceType(
                Make_TM_SimpleName(
                  mk_TM_SimpleName("Integer")),
                TM_NoOptionalReferenceType))
          else Make_TM_PrimitiveType(TM_JAVA_INT)
          end,
        TM_RSL_REAL →
          if useWrapper
          then
            Make_TM_ReferenceType(
              mk_TM_ReferenceType(
```

```
                        Make_TM_SimpleName(
                          mk_TM_SimpleName("Double")),
                        TM_NoOptionalReferenceType))
                  else Make_TM_PrimitiveType(TM_JAVA_DOUBLE)
                  end,
               TM_RSL_CHAR →
                  if useWrapper
                  then
                     Make_TM_ReferenceType(
                        mk_TM_ReferenceType(
                           Make_TM_SimpleName(
                             mk_TM_SimpleName("Character")),
                           TM_NoOptionalReferenceType))
                  else Make_TM_PrimitiveType(TM_JAVA_CHAR)
                  end,
               TM_RSL_TEXT →
                  Make_TM_ReferenceType(
                     mk_TM_ReferenceType(
                        Make_TM_SimpleName(
                           mk_TM_SimpleName("String")),
                        TM_NoOptionalReferenceType)),
               TM_RSL_BOOL →
                  if useWrapper
                  then
                     Make_TM_ReferenceType(
                        mk_TM_ReferenceType(
                           Make_TM_SimpleName(
                             mk_TM_SimpleName("Boolean")),
                           TM_NoOptionalReferenceType))
                  else Make_TM_PrimitiveType(TM_JAVA_BOOL)
                  end
            end,
         TM_TypeExprList(tel) →
            case tel of
               TM_FiniteListTypeExpr(flte) →
                  if useInterface
                  then
                     Make_TM_ReferenceType(
                        mk_TM_ReferenceType(
                           Make_TM_SimpleName(
                             mk_TM_SimpleName("RSLList")),
```

```
                                Make_TM_OptionalReferenceType(
                                  matchTypeReferenceType(
                                    flte, true, useInterface))))
                    else
                        Make_TM_ReferenceType(
                            mk_TM_ReferenceType(
                                Make_TM_SimpleName(
                                  mk_TM_SimpleName("RSLListDefault")
                                ),
                                Make_TM_OptionalReferenceType(
                                  matchTypeReferenceType(
                                    flte, true, useInterface))))
                    end
                end,
            TM_TypeName(id) →
                Make_TM_ReferenceType(
                    mk_TM_ReferenceType(
                        Make_TM_SimpleName(rslIdToSimpleName(id)),
                        TM_NoOptionalReferenceType)),
            _ →
                Make_TM_ReferenceType(
                    mk_TM_ReferenceType(
                        Make_TM_SimpleName(
                            mk_TM_SimpleName(
                                "UNKNOWN_TYPE_MATCH_TYPE")),
                        TM_NoOptionalReferenceType))
        end,

    makeName : Text* → TM_JavaName
    makeName(l) ≡
        if l = ⟨⟩
        then
            Make_TM_SimpleName(mk_TM_SimpleName("nopackage"))
        else
            if tl l = ⟨⟩
            then Make_TM_SimpleName(mk_TM_SimpleName(hd l))
            else
                Make_TM_QualifiedName(
                    mk_TM_QualifiedName(
                        makeName(tl l), mk_TM_SimpleName(hd l)))
            end
```

```
    end,

matchTypeReferenceType :
    TM_TypeEvaluator × Bool × Bool → TM_ReferenceType
matchTypeReferenceType(te, useWrapper, useInterface) ≡
    case te of
        TM_TypeEvaluator_TM_TypeExpr(t) →
            matchTypeReferenceType(
                t, useWrapper, useInterface),
        TM_TypeEvaluator_TM_Constructor(c) →
            mk_TM_ReferenceType(
                Make_TM_SimpleName(rslIdToSimpleName(id(c))),
                TM_NoOptionalReferenceType),
        _ →
            mk_TM_ReferenceType(
                Make_TM_SimpleName(
                    mk_TM_SimpleName(
                        "UNKNOWN_TYPE_MATCH_TYPE_REFERENCE")),
                TM_NoOptionalReferenceType)
    end,

matchTypeReferenceType :
    TM_TypeExpr × Bool × Bool → TM_ReferenceType
matchTypeReferenceType(te, useWrapper, useInterface) ≡
    case matchType(te, useWrapper, useInterface) of
        Make_TM_ReferenceType(rt) → rt
    end,

rslIdToSimpleName : TM_Id → TM_SimpleName
rslIdToSimpleName(id) ≡ mk_TM_SimpleName(getText(id)),

rslIdToSimpleName : TM_Pattern → TM_SimpleName
rslIdToSimpleName(p) ≡
    case p of
        TM_NamePattern(id, vi) → rslIdToSimpleName(id),
        _ →
            mk_TM_SimpleName(
                "UNKNOWN_VARIABLE_NAME_IN_PATTERN")
    end,

valueInitializerToSimpleName :
```

```
        TM_Pattern → TM_SimpleName
    valueInitializerToSimpleName(p) ≡
        case p of
            TM_NamePattern(id, vi) →
                case vi of
                    Make_TM_Id(id2) → rslIdToSimpleName(id2),
                    _ →
                        mk_TM_SimpleName(
                            "UNKNOWN_VARIABLE_INITIALIZER")
                end,
            _ →
                mk_TM_SimpleName("UNKNOWN_VARIABLE_INITIALIZER")
        end,

    textInList : Text × Text* → Bool
    textInList(t, tList) ≡
        if tList = ⟨⟩ then false
        else
            if hd tList = t then true
            else textInList(t, tl tList)
            end
        end,

    convert : Int → Text
    convert(i) ≡
        case i of
            0 → "0",
            1 → "1",
            2 → "2",
            3 → "3",
            4 → "4",
            5 → "5",
            6 → "6",
            7 → "7",
            8 → "8",
            9 → "9",
            _ → "TOO_MANY_LEVELS"
        end
end
```

C.1.2 RSLAst_Module

```
scheme RSLAst_Module =
  class
    type
      TM_RSLAst :: libmodule : TM_LibModule,
      TM_LibModule ::
        context_list : TM_Id*  schemedef : TM_SchemeDef,
      TM_SchemeDef :: id : TM_Id  class_expr : TM_ClassExpr,
      TM_ClassExpr ==
        TM_BasicClassExpr(declaration_list : TM_Decl*) |
        TM_ExtendingClassExpr(
          base_class : TM_ClassExpr,
          extension_class : TM_ClassExpr) |
        TM_SchemeInstantiation(id : TM_Id),
      TM_Decl ==
        TM_ValueDecl(value_def_list : TM_ValueDef*) |
        TM_TypeDecl(type_def_list : TM_TypeDef*) |
        TM_TestDecl(test_def_list : TM_TestDef*),
      /*Type Definitions*/
      TM_TypeDef ==
        TM_SortDef(sd_id : TM_Id) |
        TM_VariantDef(
          id : TM_Id, variant_list : TM_Variant*) |
        TM_ShortRecordDef(
          srd_id : TM_Id,
          component_kind_string : TM_ComponentKind*),
      TM_Variant ==
        Make_TM_Constructor(constructor : TM_Constructor) |
        TM_RecordVariant(
          record_constructor : TM_Constructor,
          component_kind_list : TM_ComponentKind*),
      TM_ComponentKind ::
        optional_destructor : TM_OptionalDestructor
        type_expr : TM_TypeExpr
        optional_reconstructor : TM_OptionalReconstructor,
      TM_Constructor :: id : TM_Id,
      TM_OptionalDestructor ==
        TM_Destructor(id : TM_Id) | TM_NoDestructor,
      TM_OptionalReconstructor ==
        TM_Reconstructor(id : TM_Id) | TM_NoReconstructor,
```

```
/∗Value Definitions∗/
TM_ValueDef ==
  TM_ExplicitFunctionDef(
      single_typing : TM_SingleTyping,
      formal_function_application :
        TM_FormalFunctionApplication,
      value_expr : TM_ValueExpr),
TM_SingleTyping ::
  binding : TM_Binding   type_expr : TM_TypeExpr,
TM_FormalFunctionApplication ==
  TM_IdApplication(
      id : TM_Id,
      formal_function_parameter_list :
        TM_FormalFunctionParameter*),
TM_FormalFunctionParameter ::
  binding_list : TM_Binding*,
TM_Binding :: id : TM_Id,
/∗Test Definitions∗/
TM_TestDef ==
  TM_TestCase(id : TM_Id, value_expr : TM_ValueExpr),
/∗Type Expressions∗/
TM_TypeExpr ==
  TM_TypeExprList(type_expr_list : TM_TypeExprLists) |
  TM_FunctionTypeExpr(
      type_expr_argument : TM_TypeExpr,
      function_arrow : TM_FunctionArrow,
      type_expr_result : TM_TypeExpr) |
  TM_TypeLiteral(type_literal : TM_TypeLiterals) |
  TM_TypeExprProduct(component_list : TM_TypeExpr*) |
  TM_TypeName(id : TM_Id),
TM_FunctionArrow == TM_TOTAL_FUNCTION_ARROW,
TM_TypeExprLists ==
  TM_FiniteListTypeExpr(type_expr : TM_TypeExpr),
TM_TypeLiterals ==
  TM_RSL_UNIT |
  TM_RSL_INT |
  TM_RSL_NAT |
  TM_RSL_REAL |
  TM_RSL_BOOL |
  TM_RSL_CHAR |
  TM_RSL_TEXT,
```

```
/*Value Expression*/
TM_ValueExpr ==
    Make_TM_IfExpr(if_expr : TM_IfExpr) |
    Make_TM_ValueLiteral(value_literal : TM_ValueLiteral) |
    TM_ValueInfixExpr(
        left : TM_ValueExpr,
        op : TM_InfixOperator,
        right : TM_ValueExpr) |
    TM_ValuePrefixExpr(
        op : TM_PrefixOperator, operand : TM_ValueExpr) |
    Make_TM_ListExpr(list_expr : TM_ListExpr) |
    TM_ApplicationExpr(
        value_expr : TM_ValueExpr,
        value_expr_list : TM_ValueExpr*) |
    Make_TM_ValueOrVariableName(
        value_or_variable_name : TM_ValueOrVariableName) |
    TM_ParenthesizedExpr(
        parenthesized_expr : TM_ValueExpr) |
    TM_CaseExpr(
        condition : TM_ValueExpr,
        case_branch_list : TM_CaseBranch*),
TM_ValueLiteral ==
    TM_ValueLiteralInteger(getTextInteger : Text) |
    TM_ValueLiteralReal(getTextReal : Text) |
    TM_ValueLiteralBool(getTextBool : Text) |
    TM_ValueLiteralChar(getTextChar : Text) |
    TM_ValueLiteralText(getTextText : Text),
TM_ValueOrVariableName :: id : TM_Id,
TM_IfExpr ::
    condition : TM_ValueExpr
    if_case : TM_ValueExpr
    elsif_list : TM_Elsif*
    else_case : TM_ValueExpr,
TM_Elsif ::
    condition : TM_ValueExpr   elsif_case : TM_ValueExpr,
TM_CaseBranch ::
    pattern : TM_Pattern   value_expr : TM_ValueExpr,
TM_Pattern ==
    TM_ValueLiteralPattern(
        value_literal : TM_ValueLiteral) |
    TM_NamePattern(
```

```
            id : TM_Id, value_initializer : TM_OptionalId) |
        TM_RecordPattern(
            value_or_variable_name : TM_ValueOrVariableName,
            inner_pattern_list : TM_Pattern*) |
        TM_WildcardPattern,
    TM_ListExpr ==
        TM_EnumeratedListExpr(
            value_expr_list : TM_ValueExpr*),
    TM_InfixOperator ==
        TM_RSL_PLUS | TM_RSL_EQUAL | TM_RSL_HAT | TM_RSL_STAR,
    TM_PrefixOperator == TM_RSL_HD | TM_RSL_TL,
    /*Commom*/
    TM_OptionalId ==
        Make_TM_Id(id : TM_Id) | TM_NoOptionalId,
    TM_Id :: getText : Text
end
```

C.1.3 RSLAst_WrapperModule

context: RSLAst_Module

```
scheme RSLAst_WrapperModule =
  extend RSLAst_Module with
  class
    type
      /*Type evaluation*/
      TM_TypeEvaluator ==
        TM_TypeEvaluator_TM_TypeExpr(type_expr : TM_TypeExpr) |
        TM_TypeEvaluator_TM_Constructor(
            constructor : TM_Constructor) |
        TM_TypeEvaluator_TM_Destructor(
            destructor : TM_OptionalDestructor)
  end
```

C.1.4 RSLAst_WrapperModule_2

context: RSLAst_Module, RSLAst_WrapperModule, JavaAst_Module

scheme RSLAst_WrapperModule_2 =

```
extend JavaAst_Module with
extend RSLAst_WrapperModule with
class
  value
    /*
    getTypeEvaluatorForTypeEvaluation : TM_ValueExpr
    → TM_TypeExpr
    getTypeEvaluatorForTypeEvaluation(ve) is
    TM_TypeExprList(TM_FiniteListTypeExpr(TM_TypeLiteral(
    TM_RSL_INT))),
    getTypeEvaluatorForTypeEvaluation : TM_ListExpr
    → TM_TypeExpr
    getTypeEvaluatorForTypeEvaluation(ve) is
    TM_TypeExprList(TM_FiniteListTypeExpr(TM_TypeLiteral(
    TM_RSL_INT))),
    getTypeEvaluatorForTypeEvaluation : TM_ValueOrVariableName
    → TM_TypeExpr
    getTypeEvaluatorForTypeEvaluation(vovn) is
    TM_TypeExprList(TM_FiniteListTypeExpr(TM_TypeLiteral(
    TM_RSL_INT)))
    */
    getTypeEvaluatorForTypeEvaluation :
      TM_ValueExpr → TM_TypeEvaluator
    getTypeEvaluatorForTypeEvaluation(ve) ≡
      TM_TypeEvaluator_TM_TypeExpr(
        TM_TypeExprList(
          TM_FiniteListTypeExpr(
            TM_TypeLiteral(TM_RSL_INT)))),

    getTypeEvaluatorForTypeEvaluation :
      TM_ListExpr → TM_TypeEvaluator
    getTypeEvaluatorForTypeEvaluation(ve) ≡
      TM_TypeEvaluator_TM_TypeExpr(
        TM_TypeExprList(
          TM_FiniteListTypeExpr(
            TM_TypeLiteral(TM_RSL_INT)))),

    getTypeEvaluatorForTypeEvaluation :
      TM_ValueOrVariableName → TM_TypeEvaluator
    getTypeEvaluatorForTypeEvaluation(vovn) ≡
      TM_TypeEvaluator_TM_TypeExpr(
```

```
            TM_TypeExprList(
                TM_FiniteListTypeExpr(
                    TM_TypeLiteral(TM_RSL_INT)))),

        getTypeEvaluatorForTypeEvaluation :
            TM_Pattern → TM_TypeEvaluator
        getTypeEvaluatorForTypeEvaluation(p) ≡
            TM_TypeEvaluator_TM_TypeExpr(
                TM_TypeExprList(
                    TM_FiniteListTypeExpr(
                        TM_TypeLiteral(TM_RSL_INT)))),

        Runner_instance_typeDecorate :
            TM_RSLAst → TM_CompilationUnit*
        Runner_instance_typeDecorate(ra) ≡ ⟨⟩,

        Runner_instance_translate :
            Text × Bool → TM_CompilationUnit*
        Runner_instance_translate(id, b) ≡ ⟨⟩
    end
```

C.1.5 JavaAst_Module

```
scheme JavaAst_Module =
    class
        type
            TM_JavaAst ::
                compilationUnitList : TM_CompilationUnit*,
            TM_CompilationUnit ::
                optionalPackageDeclaration :
                    TM_OptionalPackageDeclaration
                importDeclarationList : TM_ImportDeclaration*
                typeDeclarationList : TM_TypeDeclaration*,
            TM_OptionalPackageDeclaration ==
                TM_NoPackageDeclaration |
                TM_PackageDeclaration(name : TM_JavaName),
            TM_ImportDeclaration :: name : TM_JavaName,
            TM_TypeDeclaration ==
                TM_ClassDeclaration(
                    modifierList : TM_Modifier*,
```

```
        name : TM_SimpleName,
        extendName : TM_OptionalSimpleName,
        implementList : TM_SimpleName*,
        constructorDeclarationList :
          TM_ConstructorDeclaration*,
        methodDeclarationList : TM_MethodDeclaration*,
        fieldDeclarationList : TM_FieldDeclaration*,
        typeDeclarationList : TM_TypeDeclaration*),
TM_FieldDeclaration ::
  modifierList : TM_Modifier*
  getType : TM_JavaType
  variableDeclarationFragment :
    TM_VariableDeclarationFragment,
TM_ConstructorDeclaration ::
  modifierList : TM_Modifier*
  name : TM_SimpleName
  argumentList : TM_SingleVariableDeclaration*
  block : TM_Block,
TM_MethodDeclaration ::
  modifierList : TM_Modifier*
  name : TM_SimpleName
  returnType : TM_JavaType
  argumentList : TM_SingleVariableDeclaration*
  block : TM_OptionalBlock,
TM_JavaType ==
  Make_TM_PrimitiveType(
      primitiveType : TM_PrimitiveType) |
  Make_TM_ReferenceType(
      referenceType : TM_ReferenceType) |
  Make_TM_ArrayType(arrayType : TM_ArrayType),
TM_PrimitiveType ==
  TM_JAVA_INT |
  TM_JAVA_VOID |
  TM_JAVA_DOUBLE |
  TM_JAVA_BOOL |
  TM_JAVA_CHAR,
TM_ReferenceType ::
  name : TM_JavaName
  optionalTypeArgument : TM_OptionalReferenceType,
TM_ArrayType :: getType : TM_JavaType,
TM_OptionalReferenceType ==
```

```
    Make_TM_OptionalReferenceType(
        referenceType : TM_ReferenceType) |
    TM_NoOptionalReferenceType,
TM_SingleVariableDeclaration ::
    modifierList : TM_Modifier*
    getType : TM_JavaType
    name : TM_SimpleName
    optionalInitialization : TM_OptionalExpression,
TM_OptionalBlock ==
    Make_TM_OptionalBlock(block : TM_Block) |
    TM_NoOptionalBlock,
TM_Block :: statementList : TM_Statement*,
TM_OptionalStatement ==
    Make_TM_Statement(statement : TM_Statement) |
    TM_NoOptionalStatement,
TM_Statement ==
    TM_ExpressionStatement(expression : TM_Expression) |
    TM_IfStatement(
        condition : TM_Expression,
        ifBlock : TM_Block,
        elseBlock : TM_OptionalBlock) |
    TM_ReturnStatement(
        expressionReturnStatement : TM_Expression) |
    Make_TM_VariableDeclarationStatement(
        variableDeclarationStatement :
            TM_VariableDeclarationStatement),
TM_VariableDeclarationStatement ::
    modifierList : TM_Modifier*
    getType : TM_JavaType
    fragment : TM_VariableDeclarationFragment,
TM_VariableDeclarationFragment ::
    name : TM_SimpleName
    optionalExpression : TM_OptionalExpression,
TM_OptionalExpression ==
    Make_TM_OptionalExpression(
        expression : TM_Expression) |
    TM_NoOptionalExpression,
TM_Expression ==
    Make_TM_JavaValueLiteral(
        valueLiteral : TM_JavaValueLiteral) |
    Make_TM_JavaName(name : TM_JavaName) |
```

```
    TM_ArrayCreation(
        getArrayType : TM_JavaType,
        getCount : TM_OptionalExpression,
        elementList : TM_Expression*) |
    TM_InfixExpression(
        left : TM_Expression,
        op : TM_JavaInfixOperator,
        right : TM_Expression) |
    TM_PrefixExpression(
        prefixOperator : TM_JavaPrefixOperator,
        prefixExpression : TM_Expression) |
    TM_MethodInvocation(
        optionalExpression : TM_OptionalExpression,
        name : TM_SimpleName,
        argumentList : TM_Expression*) |
    TM_ClassInstanceCreation(
        optionalExpressionClassInstanceCreation :
            TM_OptionalExpression,
        getType : TM_ReferenceType,
        argumentListClassInstanceCreation :
            TM_Expression*) |
    TM_AssignmentExpression(
        lhs : TM_Expression,
        assignmentOp : TM_AssignmentOperator,
        rhs : TM_Expression) |
    TM_ParenthesizedExpression(
        expression : TM_Expression) |
    TM_InstanceOfExpression(
        instanceOfExpression : TM_Expression,
        instanceOfType : TM_JavaType) |
    TM_ThisExpression(thisName : TM_OptionalSimpleName) |
    TM_CastExpression(
        castType : TM_JavaType,
        castExpression : TM_Expression) |
    TM_FieldAccessExpression(
        fieldExpression : TM_OptionalExpression,
        fieldName : TM_SimpleName),
TM_JavaValueLiteral ==
    Make_TM_JavaValueLiteralInteger(
        valueLiteralInteger : TM_JavaValueLiteralInteger) |
    Make_TM_JavaValueLiteralString(
```

```
            valueLiteralString : TM_JavaValueLiteralString) |
        Make_TM_JavaValueLiteralDouble(
            valueLiteralDouble : TM_JavaValueLiteralDouble) |
        Make_TM_JavaValueLiteralChar(
            valueLiteralChar : TM_JavaValueLiteralChar) |
        Make_TM_JavaValueLiteralBool(
            valueLiteralBool : TM_JavaValueLiteralBool) |
        TM_NullLiteral,
    TM_JavaValueLiteralInteger :: getText : Text,
    TM_JavaValueLiteralString :: getText : Text,
    TM_JavaValueLiteralChar :: getText : Text,
    TM_JavaValueLiteralDouble :: getText : Text,
    TM_JavaValueLiteralBool :: getText : Text,
    TM_JavaInfixOperator ==
        TM_JAVA_EQUALS | TM_JAVA_PLUS | TM_JAVA_STAR,
    TM_JavaPrefixOperator == TM_JAVA_NOT,
    TM_AssignmentOperator == TM_JAVA_ASSIGNMENT_OP_EQUAL,
    TM_JavaName ==
        Make_TM_SimpleName(name : TM_SimpleName) |
        Make_TM_QualifiedName(name : TM_QualifiedName),
    TM_SimpleName :: text : Text,
    TM_QualifiedName ::
        left : TM_JavaName   right : TM_SimpleName,
    TM_OptionalSimpleName ==
        Make_TM_OptionalSimpleName(name : TM_SimpleName) |
        TM_NoOptionalSimpleName,
    TM_Modifier ==
        TM_JAVA_PUBLIC |
        TM_JAVA_STATIC |
        TM_JAVA_ABSTRACT |
        TM_JAVA_PRIVATE
end
```

C.2 Second Version

C.2.1 Translator_Module2

context: RSLAst_WrapperModule_22

```
scheme Translator_Module2 =
  extend RSLAst_WrapperModule_22 with
  class
    value
      rslAst2JavaAst :
        TM_RSLAst × TM_OptionalPackageDeclaration ×
        TM_ImportDeclaration* ×
        TM_OptionalSimpleName × TM_OptionalId ×
        Text* × Bool →
          TM_JavaAst
      rslAst2JavaAst(
        ra, opd, idl, ext, visitorId, ignoreList,
        writeExtensionFiles) ≡
        mk_TM_JavaAst(
          rslAst2CompilationUnitList(
            class_expr(schemedef(libmodule(ra))), ra, opd,
            idl, ext, visitorId, ignoreList,
            writeExtensionFiles)),

      rslAst2CompilationUnitList :
        TM_ClassExpr × TM_RSLAst ×
        TM_OptionalPackageDeclaration ×
        TM_ImportDeclaration* ×
        TM_OptionalSimpleName × TM_OptionalId ×
        Text* × Bool →
          TM_CompilationUnit*
      rslAst2CompilationUnitList(
        ce, ra, opd, idl, ext, visitorId, ignoreList,
        writeExtensionFiles) ≡
        case ce of
          TM_ExtendingClassExpr(ce1, ce2) →
            rslAst2CompilationUnitList(
              ce1, ra, opd, idl, ext, visitorId,
              ignoreList, writeExtensionFiles) ⌢
            rslAst2CompilationUnitList(
              ce2, ra, opd, idl, ext, visitorId,
              ignoreList, writeExtensionFiles),
          TM_SchemeInstantiation(id) →
            Runner_instance_translate(
              getText(id), writeExtensionFiles),
          TM_BasicClassExpr(dl) →
```

```
        if
            textInList(
                getText(id(schemedef(libmodule(ra)))),
                ignoreList)
        then ⟨⟩
        else
            Runner_instance_typeDecorate(ra) ⌢
            ⟨mk_TM_CompilationUnit(
                    opd, idl,
                    ⟨makeTypeDeclaration(
                            ce, id(schemedef(libmodule(ra))))⟩)
            ⟩ ⌢
            makeCompilationUnitList(
                ce, opd, idl, ext, visitorId)
        end
    end,

makeImportDeclarationList :
    Text* → TM_ImportDeclaration*
makeImportDeclarationList(l) ≡
    if l = ⟨⟩ then ⟨⟩
    else
        ⟨mk_TM_ImportDeclaration(
                Make_TM_SimpleName(mk_TM_SimpleName(hd l)))⟩ ⌢
        makeImportDeclarationList(tl l)
    end,

makeTypeDeclaration :
    TM_ClassExpr × TM_Id → TM_TypeDeclaration
makeTypeDeclaration(ce, id) ≡
    case ce of
        TM_BasicClassExpr(dl) →
            TM_ClassDeclaration(
                ⟨TM_JAVA_PUBLIC⟩, rslIdToSimpleName(id),
                TM_NoOptionalSimpleName, ⟨⟩, ⟨⟩,
                makeMethodDeclarationList1(dl) ⌢
                makeMainMethodDeclaration(dl), ⟨⟩, ⟨⟩),
        _ →
            TM_ClassDeclaration(
                ⟨⟩,
                mk_TM_SimpleName(
```

```
                    "NOT_HANDLED_CLASS_EXPRESSION_RECEIVED"),
                TM_NoOptionalSimpleName, ⟨⟩, ⟨⟩, ⟨⟩, ⟨
                ⟩, ⟨⟩)
    end,

makeMethodDeclarationList1 :
    TM_Decl* → TM_MethodDeclaration*
makeMethodDeclarationList1(dl) ≡
    if dl = ⟨⟩ then ⟨⟩
    else
        case hd dl of
            TM_ValueDecl(vdl) →
                makeMethodDeclarationList2(vdl) ⌢
                makeMethodDeclarationList1(tl dl),
            _ → makeMethodDeclarationList1(tl dl)
        end
    end,

makeMethodDeclarationList2 :
    TM_ValueDef* → TM_MethodDeclaration*
makeMethodDeclarationList2(vdl) ≡
    if vdl = ⟨⟩ then ⟨⟩
    else
        case hd vdl of
            TM_ExplicitFunctionDef(st, ffa, ve) →
                ⟨makeMethodDeclaration(st, ffa, ve)⟩ ⌢
                makeMethodDeclarationList2(tl vdl),
            _ → makeMethodDeclarationList2(tl vdl)
        end
    end,

makeMethodDeclaration :
    TM_SingleTyping × TM_FormalFunctionApplication ×
    TM_ValueExpr →
        TM_MethodDeclaration
makeMethodDeclaration(st, ffa, ve) ≡
    mk_TM_MethodDeclaration(
        ⟨TM_JAVA_PUBLIC, TM_JAVA_STATIC⟩,
        makeMethodName(ffa), makeReturnType(st),
        makeArgumentList(ffa, st),
        Make_TM_OptionalBlock(
```

```
            makeBlock(
                ve,
                makeReturnVariable(makeReturnType(st), 0),
                true, 0))),

makeMethodName :
    TM_FormalFunctionApplication → TM_SimpleName
makeMethodName(ffa) ≡
    case ffa of
        TM_IdApplication(id, ffpList) →
            rslIdToSimpleName(id)
    end,

makeReturnType : TM_SingleTyping → TM_JavaType
makeReturnType(st) ≡
    case type_expr(st) of
        TM_FunctionTypeExpr(tea, fa, ter) →
            matchType(ter, false, true)
    end,

makeReturnVariable :
    TM_JavaType × Int → TM_VariableDeclarationStatement
makeReturnVariable(jt, currentVariableNumber) ≡
    case jt of
        Make_TM_PrimitiveType(pt) →
            case pt of
                TM_JAVA_INT →
                    mk_TM_VariableDeclarationStatement(
                        ⟨⟩, jt,
                        mk_TM_VariableDeclarationFragment(
                            mk_TM_SimpleName(
                                "_v" ⌢
                                convert(currentVariableNumber)),
                            Make_TM_OptionalExpression(
                                Make_TM_JavaValueLiteral(
                                    Make_TM_JavaValueLiteralInteger(
                                        mk_TM_JavaValueLiteralInteger(
                                            "0")))))),
                TM_JAVA_BOOL →
                    mk_TM_VariableDeclarationStatement(
                        ⟨⟩, jt,
```

```
            mk_TM_VariableDeclarationFragment(
                mk_TM_SimpleName(
                    "_v" ^
                    convert(currentVariableNumber)),
                Make_TM_OptionalExpression(
                    Make_TM_JavaValueLiteral(
                      Make_TM_JavaValueLiteralBool(
                        mk_TM_JavaValueLiteralBool(
                          "false"))))))),
    TM_JAVA_DOUBLE →
      mk_TM_VariableDeclarationStatement(
          ⟨⟩, jt,
          mk_TM_VariableDeclarationFragment(
                mk_TM_SimpleName(
                    "_v" ^
                    convert(currentVariableNumber)),
                Make_TM_OptionalExpression(
                    Make_TM_JavaValueLiteral(
                      Make_TM_JavaValueLiteralDouble(
                        mk_TM_JavaValueLiteralDouble(
                          "0.0d"))))))),
    TM_JAVA_CHAR →
      mk_TM_VariableDeclarationStatement(
          ⟨⟩, jt,
          mk_TM_VariableDeclarationFragment(
                mk_TM_SimpleName(
                    "_v" ^
                    convert(currentVariableNumber)),
                Make_TM_OptionalExpression(
                    Make_TM_JavaValueLiteral(
                      Make_TM_JavaValueLiteralChar(
                        mk_TM_JavaValueLiteralChar(
                          " "))))))),
    _ →
      mk_TM_VariableDeclarationStatement(
          ⟨⟩, jt,
          mk_TM_VariableDeclarationFragment(
                mk_TM_SimpleName(
                    "_v" ^
                    convert(currentVariableNumber)),
                Make_TM_OptionalExpression(
```

```
                                    Make_TM_JavaValueLiteral(
                                      Make_TM_JavaValueLiteralString(
                                        mk_TM_JavaValueLiteralString(
                                          "UNKNOWN PRIMITIVE TYPE")))
                                )))
            end,
          Make_TM_ReferenceType(rt) →
            mk_TM_VariableDeclarationStatement(
              ⟨⟩, jt,
              mk_TM_VariableDeclarationFragment(
                  mk_TM_SimpleName(
                    "_v" ⌢ convert(currentVariableNumber)),
                  Make_TM_OptionalExpression(
                    Make_TM_JavaValueLiteral(TM_NullLiteral)
                  )))
    end,

  makeArgumentList :
    TM_FormalFunctionApplication × TM_SingleTyping →
      TM_SingleVariableDeclaration*
  makeArgumentList(ffa, st) ≡
    case ffa of
      TM_IdApplication(id, ffpList) →
        case type_expr(st) of
          TM_FunctionTypeExpr(tea, fa, ter) →
            case tea of
              TM_TypeExprList(tel) →
                ⟨mk_TM_SingleVariableDeclaration(
                      ⟨⟩, matchType(tea, false, true),
                      rslIdToSimpleName(
                        id(hd binding_list(hd ffpList))),
                      TM_NoOptionalExpression)⟩,
              TM_TypeLiteral(tpl) →
                case tpl of
                  TM_RSL_UNIT → ⟨⟩,
                  _ →
                    ⟨mk_TM_SingleVariableDeclaration(
                      ⟨⟩,
                      matchType(tea, false, true),
                      rslIdToSimpleName(
                        id(hd binding_list(
```

```
                                        hd ffpList))),
                                TM_NoOptionalExpression)⟩
                    end,
                TM_TypeExprProduct(te_list) →
                    makeArgumentListTypeExprProduct(
                        binding_list(hd ffpList), te_list),
                TM_TypeName(tn) →
                    ⟨mk_TM_SingleVariableDeclaration(
                            ⟨⟩, matchType(tea, false, true),
                            rslIdToSimpleName(
                              id(hd binding_list(hd ffpList))),
                            TM_NoOptionalExpression)⟩
            end
        end
    end,

makeArgumentListTypeExprProduct :
    TM_Binding* × TM_TypeExpr* →
        TM_SingleVariableDeclaration*
makeArgumentListTypeExprProduct(bl, tel) ≡
    if bl = ⟨⟩ then ⟨⟩
    else
        ⟨mk_TM_SingleVariableDeclaration(
                ⟨⟩, matchType(hd tel, false, true),
                rslIdToSimpleName(id(hd bl)),
                TM_NoOptionalExpression)⟩ ⌢
        makeArgumentListTypeExprProduct(tl bl, tl tel)
    end,

makeArgumentList :
    TM_ComponentKind* × Int →
        TM_SingleVariableDeclaration*
makeArgumentList(ckl, i) ≡
    if ckl = ⟨⟩ then ⟨⟩
    else
        ⟨mk_TM_SingleVariableDeclaration(
                ⟨⟩,
                matchType(type_expr(hd ckl), false, true),
                mk_TM_SimpleName("_v" ⌢ convert(i)),
                TM_NoOptionalExpression)⟩ ⌢
        makeArgumentList(tl ckl, i + 1)
```

```
    end,

makeBlock :
    TM_ValueExpr × TM_VariableDeclarationStatement ×
    Bool × Int →
        TM_Block
makeBlock(ve, vds, returnValue, currentVariableNumber) ≡
    if returnValue
    then
        mk_TM_Block(
            ⟨Make_TM_VariableDeclarationStatement(vds)⟩ ⌢
            makeStatementList(
                  ve,
                  makeReturnVariable(
                       getType(vds), currentVariableNumber),
                  currentVariableNumber) ⌢
            ⟨TM_ReturnStatement(
                    Make_TM_JavaName(
                        Make_TM_SimpleName(name(fragment(vds))))
               )⟩)
    else
        mk_TM_Block(
            ⟨Make_TM_VariableDeclarationStatement(
                    makeReturnVariable(
                        getType(vds), currentVariableNumber))⟩ ⌢
            makeStatementList(
                  ve,
                  makeReturnVariable(
                       getType(vds), currentVariableNumber),
                  currentVariableNumber) ⌢
            ⟨TM_ExpressionStatement(
                    TM_AssignmentExpression(
                        Make_TM_JavaName(
                              Make_TM_SimpleName(
                                name(fragment(vds)))),
                        TM_JAVA_ASSIGNMENT_OP_EQUAL,
                        Make_TM_JavaName(
                              Make_TM_SimpleName(
                                name(
                                  fragment(
                                    makeReturnVariable(
```

```
                                        getType(vds),
                                        currentVariableNumber))))
                        )))))
    end,

makeStatementList :
    TM_ValueExpr × TM_VariableDeclarationStatement ×
    Int →
        TM_Statement*
makeStatementList(
     valueExpr, vds, currentVariableNumber) ≡
    case valueExpr of
        Make_TM_IfExpr(ifExpr) →
            makeStatementList(
                ifExpr, vds, currentVariableNumber),
        TM_CaseExpr(c, cbl) →
            makeStatementListCaseExpr(
                c, cbl, vds, currentVariableNumber),
        /*TM_CaseExpr(c, cbl) → ⟨⟩,*/
        _ →
            ⟨TM_ExpressionStatement(
                    TM_AssignmentExpression(
                        Make_TM_JavaName(
                            Make_TM_SimpleName(
                                name(fragment(vds)))),
                        TM_JAVA_ASSIGNMENT_OP_EQUAL,
                        makeExpression(valueExpr)))⟩
    end,

makeStatementList :
    TM_IfExpr × TM_VariableDeclarationStatement × Int →
        TM_Statement*
makeStatementList(ifExpr, vds, currentVariableNumber) ≡
    ⟨TM_IfStatement(
            makeExpression(condition(ifExpr)),
            makeBlock(
                if_case(ifExpr), vds, false,
                currentVariableNumber + 1),
            Make_TM_OptionalBlock(
                makeBlock(
                    else_case(ifExpr), vds, false,
```

```
                    currentVariableNumber + 1))))⟩,

makeStatementListCaseExpr :
   TM_ValueExpr × TM_CaseBranch* ×
   TM_VariableDeclarationStatement × Int →
      TM_Statement*
makeStatementListCaseExpr(
    c, cbl, vds, currentVariableNumber) ≡
   makeStatementListCaseExpr(
       cbl, makeExpression(c), vds,
       currentVariableNumber + 1, TM_NoOptionalStatement),

makeStatementListCaseExpr :
   TM_CaseBranch* × TM_Expression ×
   TM_VariableDeclarationStatement × Int ×
   TM_OptionalStatement →
      TM_Statement*
makeStatementListCaseExpr(
    cbl, e, vds, currentVariableNumber, os) ≡
   if cbl = ⟨⟩
   then
      case os of
         Make_TM_Statement(s) → ⟨s⟩,
         TM_NoOptionalStatement → ⟨⟩
      end
   else
      makeStatementListCaseExpr(
          tl cbl, e, vds, currentVariableNumber,
          Make_TM_Statement(
                makeStatement(
                    hd cbl, e, vds, currentVariableNumber, os)
          ))
   end,

makeStatement :
   TM_CaseBranch × TM_Expression ×
   TM_VariableDeclarationStatement × Int ×
   TM_OptionalStatement →
      TM_Statement
makeStatement(cb, e, vds, currentVariableNumber, os) ≡
   case pattern(cb) of
```

```
TM_NamePattern(id, vi) →
  case os of
    Make_TM_Statement(s) →
      case s of
        TM_IfStatement(
            condition, if_block, else_block) →
          case else_block of
            TM_NoOptionalBlock →
              TM_IfStatement(
                condition, if_block,
                Make_TM_OptionalBlock(
                  mk_TM_Block(
                    ⟨TM_IfStatement(
                        TM_InstanceOfExpression(
                          e,
                          Make_TM_ReferenceType(
                            mk_TM_ReferenceType
                            (
                     Make_TM_SimpleName(
                       rslIdToSimpleName(
                         id)),
                     TM_NoOptionalReferenceType))),
                            makeBlock
                            (
                      value_expr(cb),
                      vds, false,
                      currentVariableNumber),
                            TM_NoOptionalBlock)⟩)))
              ),
            Make_TM_OptionalBlock(b) →
              TM_IfStatement(
                condition, if_block,
                Make_TM_OptionalBlock(
                  mk_TM_Block(
                    ⟨makeStatement(
                        cb, e, vds,
                        currentVariableNumber,
                        Make_TM_Statement(
                          hd statementList(b))
                      )⟩)))
          end,
```

```
                _ →
                    TM_ExpressionStatement(
                        TM_ClassInstanceCreation(
                            TM_NoOptionalExpression,
                            mk_TM_ReferenceType(
                              Make_TM_SimpleName(
                                mk_TM_SimpleName("String")),
                              TM_NoOptionalReferenceType),
                            ⟨Make_TM_JavaValueLiteral(
                                Make_TM_JavaValueLiteralString(
                                  mk_TM_JavaValueLiteralString
                                  (
"SOMETHING IS REALLY WRONG")))))))
            end,
          TM_NoOptionalStatement →
            TM_IfStatement(
                TM_InstanceOfExpression(
                    e,
                    Make_TM_ReferenceType(
                        mk_TM_ReferenceType(
                          Make_TM_SimpleName(
                            rslIdToSimpleName(id)),
                          TM_NoOptionalReferenceType))),
                makeBlock(
                    value_expr(cb), vds, false,
                    currentVariableNumber),
                TM_NoOptionalBlock)
      end,
    TM_RecordPattern(vovn, ipl) →
      case os of
        Make_TM_Statement(s) →
          case s of
            TM_IfStatement(
                condition, if_block, else_block) →
              case else_block of
                TM_NoOptionalBlock →
                  TM_IfStatement(
                    condition, if_block,
                    Make_TM_OptionalBlock(
                      mk_TM_Block(
                        ⟨TM_IfStatement(
```

```
                    TM_InstanceOfExpression(
                      e,
                      Make_TM_ReferenceType(
                        mk_TM_ReferenceType
                        (
                  Make_TM_SimpleName(
                    rslIdToSimpleName(
                      id(vovn)
                    )),
                  TM_NoOptionalReferenceType))),
                        mk_TM_Block(
                          makeStatementList(
                            e, pattern(cb),
                            ipl) ⌢
                          statementList(
                            makeBlock
                            (
                          value_expr(cb),
                          vds, false,
                          currentVariableNumber))
                          ), TM_NoOptionalBlock)
                      ⟩))),
          Make_TM_OptionalBlock(b) →
            TM_IfStatement(
              condition, if_block,
              Make_TM_OptionalBlock(
                mk_TM_Block(
                  ⟨makeStatement(
                      cb, e, vds,
                      currentVariableNumber,
                      Make_TM_Statement(
                        hd statementList(b))
                    )⟩)))
    end,
_ →
    TM_ExpressionStatement(
      TM_ClassInstanceCreation(
        TM_NoOptionalExpression,
        mk_TM_ReferenceType(
          Make_TM_SimpleName(
            mk_TM_SimpleName("String")),
```

```
                                    TM_NoOptionalReferenceType),
                                ⟨Make_TM_JavaValueLiteral(
                                    Make_TM_JavaValueLiteralString(
                                      mk_TM_JavaValueLiteralString
                                      (
"SOMETHING IS REALLY WRONG")))))⟩
            end,
          TM_NoOptionalStatement →
            TM_IfStatement(
                TM_InstanceOfExpression(
                      e,
                      Make_TM_ReferenceType(
                          mk_TM_ReferenceType(
                            Make_TM_SimpleName(
                              rslIdToSimpleName(id(vovn))),
                            TM_NoOptionalReferenceType))),
                  mk_TM_Block(
                      makeStatementList(e, pattern(cb), ipl) ⌢
                      statementList(
                          makeBlock(
                            value_expr(cb), vds, false,
                            currentVariableNumber))),
                  TM_NoOptionalBlock)
      end,
    TM_ValueLiteralPattern(vl) →
      case os of
        Make_TM_Statement(s) →
          case s of
            TM_IfStatement(
                condition, if_block, else_block) →
              case else_block of
                TM_NoOptionalBlock →
                  TM_IfStatement(
                    condition, if_block,
                    Make_TM_OptionalBlock(
                      mk_TM_Block(
                        ⟨TM_IfStatement(
                            TM_InfixExpression(
                              e, TM_JAVA_EQUALS,
                              makeExpression(
                                Make_TM_ValueLiteral(
```

```
                                        vl))),
                                    makeBlock
                                    (
                                value_expr(cb),
                                vds, false,
                                currentVariableNumber),
                                    TM_NoOptionalBlock)⟩)))
                            ),
                        Make_TM_OptionalBlock(b) →
                            TM_IfStatement(
                              condition, if_block,
                              Make_TM_OptionalBlock(
                                mk_TM_Block(
                                  ⟨makeStatement(
                                    cb, e, vds,
                                    currentVariableNumber,
                                    Make_TM_Statement(
                                      hd statementList(b))
                                  )⟩⟩⟩⟩)
                    end,
                _ →
                    TM_ExpressionStatement(
                        TM_ClassInstanceCreation(
                            TM_NoOptionalExpression,
                            mk_TM_ReferenceType(
                              Make_TM_SimpleName(
                                mk_TM_SimpleName("String")),
                              TM_NoOptionalReferenceType),
                            ⟨Make_TM_JavaValueLiteral(
                                Make_TM_JavaValueLiteralString(
                                  mk_TM_JavaValueLiteralString
                                  (
"SOMETHING IS REALLY WRONG")))))))
            end,
        TM_NoOptionalStatement →
            TM_IfStatement(
                TM_InfixExpression(
                    e, TM_JAVA_EQUALS,
                    makeExpression(
                        Make_TM_ValueLiteral(vl))),
                makeBlock(
```

```
                    value_expr(cb), vds, false,
                    currentVariableNumber),
                TM_NoOptionalBlock)
    end,
TM_WildcardPattern →
    case os of
        Make_TM_Statement(s) →
            case s of
                TM_IfStatement(
                    condition, if_block, else_block) →
                    case else_block of
                        TM_NoOptionalBlock →
                            TM_IfStatement(
                                condition, if_block,
                                Make_TM_OptionalBlock(
                                    makeBlock(
                                        value_expr(cb), vds, false,
                                        currentVariableNumber))),
                        Make_TM_OptionalBlock(b) →
                            TM_IfStatement(
                                condition, if_block,
                                Make_TM_OptionalBlock(
                                    mk_TM_Block(
                                        ⟨makeStatement(
                                            cb, e, vds,
                                            currentVariableNumber,
                                            Make_TM_Statement(
                                                hd statementList(b))
                                        )⟩))))
                    end,
                _ →
                    TM_ExpressionStatement(
                        TM_ClassInstanceCreation(
                            TM_NoOptionalExpression,
                            mk_TM_ReferenceType(
                                Make_TM_SimpleName(
                                    mk_TM_SimpleName("String")),
                                TM_NoOptionalReferenceType),
                            ⟨Make_TM_JavaValueLiteral(
                                Make_TM_JavaValueLiteralString(
                                    mk_TM_JavaValueLiteralString
```

```
                                    (
    "SOMETHING IS REALLY WRONG")))⟩))
                end,
            TM_NoOptionalStatement →
                TM_ExpressionStatement(
                    makeExpression(value_expr(cb)))
        end,
    _ →
        TM_ExpressionStatement(
            TM_ClassInstanceCreation(
                TM_NoOptionalExpression,
                mk_TM_ReferenceType(
                    Make_TM_SimpleName(
                        mk_TM_SimpleName("String")),
                    TM_NoOptionalReferenceType),
                ⟨Make_TM_JavaValueLiteral(
                    Make_TM_JavaValueLiteralString(
                        mk_TM_JavaValueLiteralString(
                            "NOT_NAMEPATTERN")))⟩))
    end,

makeStatementList :
    TM_Expression × TM_Pattern × TM_Pattern* →
        TM_Statement*
makeStatementList(
    expression, recordPattern, innerPatternList) ≡
    case recordPattern of
        TM_RecordPattern(vovn, il) →
            if innerPatternList = ⟨⟩ then ⟨⟩
            else
                ⟨Make_TM_VariableDeclarationStatement(
                    mk_TM_VariableDeclarationStatement(
                        ⟨⟩,
                        matchType(
                            getTypeEvaluatorForTypeEvaluation(
                                hd innerPatternList), false, true
                        ),
                        mk_TM_VariableDeclarationFragment(
                            rslIdToSimpleName(
                                hd innerPatternList),
                            Make_TM_OptionalExpression(
```

```
                                    TM_MethodInvocation(
                                      Make_TM_OptionalExpression(
                                        TM_ParenthesizedExpression(
                                          TM_CastExpression(
                                            Make_TM_ReferenceType(
                                              mk_TM_ReferenceType
                                              (
                                    Make_TM_SimpleName(
                                      rslIdToSimpleName(
                                        id(vovn))
                                    ),
                                    TM_NoOptionalReferenceType)),
                                                expression))),
                                      valueInitializerToSimpleName(
                                        hd innerPatternList), ⟨⟩)
                                ))))⟩ ⌢
                        makeStatementList(
                            expression, recordPattern,
                            tl innerPatternList)
                    end,
                _ → ⟨⟩
            end,

        /*
        makeExpression : TM_ValueExpr → TM_Expression
        makeExpression(ve) is
        Make_TM_JavaValueLiteral(Make_TM_JavaValueLiteralInteger(
        mk_TM_JavaValueLiteralInteger("UNKNOWN_EXPRESSION")
        )),
        */
        makeExpression : TM_ValueExpr → TM_Expression
        makeExpression(ve) ≡
            case ve of
                Make_TM_ValueLiteral(vl) →
                    case vl of
                        TM_ValueLiteralInteger(t) →
                            Make_TM_JavaValueLiteral(
                                Make_TM_JavaValueLiteralInteger(
                                    mk_TM_JavaValueLiteralInteger(t))),
                        TM_ValueLiteralReal(t) →
                            Make_TM_JavaValueLiteral(
```

```
                Make_TM_JavaValueLiteralDouble(
                    mk_TM_JavaValueLiteralDouble(t))),
            TM_ValueLiteralBool(t) →
              Make_TM_JavaValueLiteral(
                Make_TM_JavaValueLiteralBool(
                    mk_TM_JavaValueLiteralBool(t))),
            TM_ValueLiteralText(t) →
              Make_TM_JavaValueLiteral(
                Make_TM_JavaValueLiteralString(
                    mk_TM_JavaValueLiteralString(t))),
            TM_ValueLiteralChar(t) →
              Make_TM_JavaValueLiteral(
                Make_TM_JavaValueLiteralChar(
                    mk_TM_JavaValueLiteralChar(t)))
        end,
    TM_ValueInfixExpr(lo, operator, ro) →
        case operator of
            TM_RSL_PLUS →
              TM_InfixExpression(
                makeExpression(lo), TM_JAVA_PLUS,
                makeExpression(ro)),
            TM_RSL_STAR →
              TM_InfixExpression(
                makeExpression(lo), TM_JAVA_STAR,
                makeExpression(ro)),
            TM_RSL_SLASH →
              TM_InfixExpression(
                makeExpression(lo), TM_JAVA_DIV,
                makeExpression(ro)),
            TM_RSL_HAT →
              TM_MethodInvocation(
                Make_TM_OptionalExpression(
                    makeExpression(lo)),
                mk_TM_SimpleName("concat"),
                ⟨makeExpression(ro)⟩),
            TM_RSL_EQUAL →
              case
                matchType(
                  getTypeEvaluatorForTypeEvaluation(lo),
                  false, false)
              of
```

```
                Make_TM_PrimitiveType(pt) →
                  TM_InfixExpression(
                      makeExpression(lo), TM_JAVA_EQUALS,
                      makeExpression(ro)),
                Make_TM_ReferenceType(rt) →
                  TM_MethodInvocation(
                      Make_TM_OptionalExpression(
                          makeExpression(lo)),
                      mk_TM_SimpleName("equals"),
                      ⟨makeExpression(ro)⟩)
              end,
            _ →
              Make_TM_JavaValueLiteral(
                  Make_TM_JavaValueLiteralInteger(
                      mk_TM_JavaValueLiteralInteger(
                          "UNKNOWN_OPERATOR")))
          end,
        TM_ValuePrefixExpr(po, value_expr) →
          case po of
            TM_RSL_HD →
              TM_MethodInvocation(
                  Make_TM_OptionalExpression(
                      makeExpression(value_expr)),
                  mk_TM_SimpleName("hd"), ⟨⟩),
            TM_RSL_TL →
              TM_MethodInvocation(
                  Make_TM_OptionalExpression(
                      makeExpression(value_expr)),
                  mk_TM_SimpleName("tl"), ⟨⟩),
            _ →
              Make_TM_JavaName(
                  Make_TM_SimpleName(
                      mk_TM_SimpleName("UNKNOWN_OPERATOR")))
          end,
        TM_ApplicationExpr(value_expr, vel) →
          case
            getTypeEvaluatorForTypeEvaluation(value_expr)
          of
            TM_TypeEvaluator_TM_TypeExpr(te) →
              case te of
                TM_TypeName(tn) →
```

```
                    TM_ClassInstanceCreation(
                        TM_NoOptionalExpression,
                        mk_TM_ReferenceType(
                            Make_TM_SimpleName(
                              rslIdToSimpleName(tn)),
                            TM_NoOptionalReferenceType),
                        makeArgumentList(vel)),
                _ →
                    TM_MethodInvocation(
                        TM_NoOptionalExpression,
                        makeMethodName(value_expr),
                        makeArgumentList(vel))
            end,
        TM_TypeEvaluator_TM_Constructor(c) →
            TM_ClassInstanceCreation(
                TM_NoOptionalExpression,
                mk_TM_ReferenceType(
                    Make_TM_SimpleName(
                        rslIdToSimpleName(id(c))),
                    TM_NoOptionalReferenceType),
                makeArgumentList(vel)),
        TM_TypeEvaluator_TM_Destructor(d) →
            TM_MethodInvocation(
                Make_TM_OptionalExpression(
                    TM_ParenthesizedExpression(
                        makeExpression(hd vel))),
                makeMethodName(value_expr), ⟨⟩),
        _ →
            TM_MethodInvocation(
                TM_NoOptionalExpression,
                makeMethodName(value_expr),
                makeArgumentList(vel))
    end,
TM_ParenthesizedExpr(value_expr) →
    TM_ParenthesizedExpression(
        makeExpression(value_expr)),
Make_TM_ListExpr(ele) →
    case ele of
        TM_EnumeratedListExpr(vel) →
            if vel = ⟨⟩
            then
```

```
                TM_ClassInstanceCreation(
                    TM_NoOptionalExpression,
                    matchTypeReferenceType(
                        getTypeEvaluatorForTypeEvaluation(
                            ele), true, false), ⟨⟩)
            else
                TM_ClassInstanceCreation(
                    TM_NoOptionalExpression,
                    matchTypeReferenceType(
                        getTypeEvaluatorForTypeEvaluation(
                            ele), true, false),
                    ⟨TM_ArrayCreation(
                        matchTypeInner(
                            getTypeEvaluatorForTypeEvaluation(
                                ele), true, false),
                        TM_NoOptionalExpression,
                        makeArgumentList(vel))⟩)
            end,
        _ →
            Make_TM_JavaName(
                Make_TM_SimpleName(
                    mk_TM_SimpleName("UNKNOWN_LIST_EXPR")))
    end,
Make_TM_ValueOrVariableName(vovn) →
    case getTypeEvaluatorForTypeEvaluation(vovn) of
        TM_TypeEvaluator_TM_Constructor(c) →
            TM_ClassInstanceCreation(
                TM_NoOptionalExpression,
                mk_TM_ReferenceType(
                    Make_TM_SimpleName(
                        rslIdToSimpleName(id(c))),
                    TM_NoOptionalReferenceType), ⟨⟩),
        /*TM_TypeEvaluator_TM_TypeExpr(te) → Make_TM_
        JavaName(Make_TM_SimpleName(mk_TM_SimpleName(
        getText(id(vovn))))),*/
        _ →
            Make_TM_JavaName(
                Make_TM_SimpleName(
                    mk_TM_SimpleName(getText(id(vovn)))))
        /*_ → Make_TM_JavaValueLiteral(Make_TM_JavaValueLiteralInteA
        ger(mk_TM_JavaValueLiteralInteger("UNKNOWN_VALUE_
```

```
            OR_VARIABLE_NAME")))*/
            end,
        _ →
            Make_TM_JavaValueLiteral(
                Make_TM_JavaValueLiteralInteger(
                    mk_TM_JavaValueLiteralInteger(
                        "UNKNOWN_EXPRESSION")))
    end,

makeMethodName : TM_ValueExpr → TM_SimpleName
makeMethodName(ve) ≡
    case ve of
        Make_TM_ValueOrVariableName(vovn) →
            mk_TM_SimpleName(getText(id(vovn))),
        _ → mk_TM_SimpleName("UNKNOWN_METHOD_NAME")
    end,

makeArgumentList :
    TM_ValueExpr* → TM_Expression*
makeArgumentList(vel) ≡
    if vel = ⟨⟩ then ⟨⟩
    else
        ⟨makeExpression(hd vel)⟩ ⌢
        makeArgumentList(tl vel)
    end,

makeMainMethodDeclaration :
    TM_Decl* → TM_MethodDeclaration*
makeMainMethodDeclaration(dl) ≡
    if makeMainMethodDeclaration1(dl) = ⟨⟩ then ⟨⟩
    else
        ⟨mk_TM_MethodDeclaration(
                ⟨TM_JAVA_PUBLIC, TM_JAVA_STATIC⟩,
                mk_TM_SimpleName("main"),
                Make_TM_PrimitiveType(TM_JAVA_VOID),
                ⟨mk_TM_SingleVariableDeclaration(
                        ⟨⟩,
                        Make_TM_ArrayType(
                            mk_TM_ArrayType(
                                Make_TM_ReferenceType(
                                  mk_TM_ReferenceType(
```

```
                                        Make_TM_SimpleName(
                                          mk_TM_SimpleName("String")
                                        ), TM_NoOptionalReferenceType
                                      )))), mk_TM_SimpleName("args"),
                          TM_NoOptionalExpression)⟩,
                Make_TM_OptionalBlock(
                      mk_TM_Block(makeMainMethodDeclaration1(dl))
                ))⟩
    end,

makeMainMethodDeclaration1 :
    TM_Decl* → TM_Statement*
makeMainMethodDeclaration1(dl) ≡
    if dl = ⟨⟩ then ⟨⟩
    else
        case hd dl of
            TM_TestDecl(tdl) →
                makeMainMethodDeclaration2(tdl) ⌢
                makeMainMethodDeclaration1(tl dl),
            _ → makeMainMethodDeclaration1(tl dl)
        end
    end,

makeMainMethodDeclaration2 :
    TM_TestDef* → TM_Statement*
makeMainMethodDeclaration2(tdl) ≡
    if tdl = ⟨⟩ then ⟨⟩
    else
        case hd tdl of
            TM_TestCase(id, value_expr) →
                makeTestDeclaration(id, value_expr) ⌢
                makeMainMethodDeclaration2(tl tdl),
            _ → makeMainMethodDeclaration2(tl tdl)
        end
    end,

makeTestDeclaration :
    TM_Id × TM_ValueExpr → TM_Statement*
makeTestDeclaration(id, ve) ≡
    ⟨TM_ExpressionStatement(
          TM_MethodInvocation(
```

```
                Make_TM_OptionalExpression(
                    Make_TM_JavaName(
                        Make_TM_QualifiedName(
                            mk_TM_QualifiedName(
                                Make_TM_SimpleName(
                                    mk_TM_SimpleName("System")),
                                mk_TM_SimpleName("out")))))),
                mk_TM_SimpleName("println"),
                ⟨TM_InfixExpression(
                    Make_TM_JavaValueLiteral(
                        Make_TM_JavaValueLiteralString(
                            mk_TM_JavaValueLiteralString(
                                "[" ⌢ getText(id) ⌢ "]: "))),
                    TM_JAVA_PLUS, makeExpression(ve))⟩⟩⟩⟩,

makeCompilationUnitList :
    TM_ClassExpr × TM_OptionalPackageDeclaration ×
    TM_ImportDeclaration* ×
    TM_OptionalSimpleName × TM_OptionalId →
        TM_CompilationUnit*
makeCompilationUnitList(ce, opd, idl, ext, visitor) ≡
    case ce of
        TM_BasicClassExpr(dl) →
            makeCompilationUnitList1(
                dl, opd, idl, ext, visitor),
        _ → ⟨⟩
    end,

makeCompilationUnitList1 :
    TM_Decl* × TM_OptionalPackageDeclaration ×
    TM_ImportDeclaration* ×
    TM_OptionalSimpleName × TM_OptionalId →
        TM_CompilationUnit*
makeCompilationUnitList1(dl, opd, idl, ext, visitor) ≡
    if dl = ⟨⟩ then ⟨⟩
    else
        case hd dl of
            TM_TypeDecl(tdl) →
                makeCompilationUnitList2(
                    tdl, opd, idl, ext, visitor) ⌢
                makeCompilationUnitList1(
```

```
                tl dl, opd, idl, ext, visitor),
        _ →
            makeCompilationUnitList1(
                tl dl, opd, idl, ext, visitor)
    end
end,

makeCompilationUnitList2 :
    TM_TypeDef* × TM_OptionalPackageDeclaration ×
    TM_ImportDeclaration* ×
    TM_OptionalSimpleName × TM_OptionalId →
        TM_CompilationUnit*
makeCompilationUnitList2(tdl, opd, idl, ext, visitor) ≡
    if tdl = ⟨⟩ then ⟨⟩
    else
        makeClassDeclaration1(
            hd tdl, opd, idl, ext, visitor) ⌢
        makeCompilationUnitList2(
            tl tdl, opd, idl, ext, visitor)
    end,

makeClassDeclaration1 :
    TM_TypeDef × TM_OptionalPackageDeclaration ×
    TM_ImportDeclaration* ×
    TM_OptionalSimpleName × TM_OptionalId →
        TM_CompilationUnit*
makeClassDeclaration1(td, opd, idl, ext, visitor) ≡
    case td of
        TM_VariantDef(id, vl) →
            ⟨makeClassDeclaration2(
                    id, opd, idl, ext, visitor)⟩ ⌢
            makeClassDeclarationList(
                vl, id, opd, idl,
                Make_TM_OptionalSimpleName(
                    rslIdToSimpleName(id)), visitor),
        TM_ShortRecordDef(id, cks) →
            case visitor of
                Make_TM_Id(visitorId) →
                    ⟨mk_TM_CompilationUnit(
                            opd, idl,
                            ⟨TM_ClassDeclaration(
```

```
                    ⟨TM_JAVA_PUBLIC⟩,
                    rslIdToSimpleName(id), ext, ⟨⟩,
                    makeConstructorDeclaration(id, cks),
                    makeStandardMethodDeclarationList(
                      id, cks) ⌢
                    makeVisitorMethod(
                      id, visitorId, false),
                    makeFieldDeclaration(cks, 0), ⟨⟩)
                ⟩)⟩,
        TM_NoOptionalId →
          ⟨mk_TM_CompilationUnit(
                opd, idl,
                ⟨TM_ClassDeclaration(
                    ⟨TM_JAVA_PUBLIC⟩,
                    rslIdToSimpleName(id), ext, ⟨⟩,
                    makeConstructorDeclaration(id, cks),
                    makeStandardMethodDeclarationList(
                      id, cks),
                    makeFieldDeclaration(cks, 0), ⟨⟩)
                ⟩)⟩
    end,
TM_SortDef(id) →
    case visitor of
      Make_TM_Id(visitorId) →
        ⟨mk_TM_CompilationUnit(
                opd, idl,
                ⟨TM_ClassDeclaration(
                    ⟨TM_JAVA_PUBLIC⟩,
                    rslIdToSimpleName(id), ext, ⟨⟩,
                    ⟨⟩,
                    ⟨mk_TM_MethodDeclaration(
                        ⟨TM_JAVA_PUBLIC⟩,
                        mk_TM_SimpleName("equals"),
                        Make_TM_PrimitiveType(
                          TM_JAVA_BOOL),
                        ⟨mk_TM_SingleVariableDeclaration(
                            ⟨⟩,
                            Make_TM_ReferenceType(
                              mk_TM_ReferenceType
                              (
                    Make_TM_SimpleName(
```

```
                    mk_TM_SimpleName(
                      "Object")),
                TM_NoOptionalReferenceType)),
                        mk_TM_SimpleName("o"),
                        TM_NoOptionalExpression)
                    ⟩,
                    Make_TM_OptionalBlock(
                      mk_TM_Block(
                        ⟨TM_ReturnStatement(
                            TM_InstanceOfExpression(
                              Make_TM_JavaName(
                                Make_TM_SimpleName(
                                  mk_TM_SimpleName(
                                    "o"))),
                              Make_TM_ReferenceType(
                                mk_TM_ReferenceType
                                (
                             Make_TM_SimpleName(
                               mk_TM_SimpleName(
                                 getText
                                 (
                               id))),
                             TM_NoOptionalReferenceType)))
                                 )⟩)))),
                    mk_TM_MethodDeclaration(
                      ⟨TM_JAVA_PUBLIC⟩,
                      mk_TM_SimpleName("toString"),
                      Make_TM_ReferenceType(
                        mk_TM_ReferenceType
                        (
Make_TM_SimpleName(
    mk_TM_SimpleName(
        "String")),
TM_NoOptionalReferenceType)), ⟨⟩,
                        Make_TM_OptionalBlock(
                          mk_TM_Block(
                            ⟨TM_ReturnStatement(
                                Make_TM_JavaValueLiteral(
                                  Make_TM_JavaValueLiteralString(
                                    mk_TM_JavaValueLiteralString(
                                      getText(
```

```
                                                    id)))))
                                        ⟩))),
                                mk_TM_MethodDeclaration(
                                  ⟨TM_JAVA_PUBLIC⟩,
                                  mk_TM_SimpleName("accept"),
                                  Make_TM_PrimitiveType(
                                    TM_JAVA_BOOL),
                                  ⟨mk_TM_SingleVariableDeclaration(
                                      ⟨⟩,
                                      Make_TM_ReferenceType(
                                        mk_TM_ReferenceType
                                        (
                          Make_TM_SimpleName(
                              rslIdToSimpleName(
                                visitorId)),
                          TM_NoOptionalReferenceType)),
                                      mk_TM_SimpleName(
                                        "visitor"),
                                      TM_NoOptionalExpression)
                                  ⟩,
                                  Make_TM_OptionalBlock(
                                    mk_TM_Block(
                                      ⟨TM_ExpressionStatement(
                                          TM_MethodInvocation(
                                            Make_TM_OptionalExpression(
                                              Make_TM_JavaName(
                                                Make_TM_SimpleName(
                                                  mk_TM_SimpleName
                                                  (
                                          "visitor")))),
                                            mk_TM_SimpleName(
                                              "visit" ^
                                              getText(id)),
                                            ⟨TM_ThisExpression
                                              (
                                    TM_NoOptionalSimpleName)⟩))
                                      ⟩))))⟩, ⟨⟩, ⟨⟩)⟩)
    ⟩,
TM_NoOptionalId →
    ⟨mk_TM_CompilationUnit(
            opd, idl,
```

```
⟨TM_ClassDeclaration(
      ⟨TM_JAVA_PUBLIC⟩,
      rslIdToSimpleName(id), ext, ⟨⟩,
      ⟨⟩,
      ⟨mk_TM_MethodDeclaration(
          ⟨TM_JAVA_PUBLIC⟩,
          mk_TM_SimpleName("equals"),
          Make_TM_PrimitiveType(
            TM_JAVA_BOOL),
          ⟨mk_TM_SingleVariableDeclaration(
              ⟨⟩,
              Make_TM_ReferenceType(
                mk_TM_ReferenceType
                (
  Make_TM_SimpleName(
      mk_TM_SimpleName(
        "Object")),
  TM_NoOptionalReferenceType)),
                mk_TM_SimpleName("o"),
                TM_NoOptionalExpression)
          ⟩,
          Make_TM_OptionalBlock(
            mk_TM_Block(
              ⟨TM_ReturnStatement(
                  TM_InstanceOfExpression(
                    Make_TM_JavaName(
                      Make_TM_SimpleName(
                        mk_TM_SimpleName(
                          "o"))),
                    Make_TM_ReferenceType(
                      mk_TM_ReferenceType
                      (
                Make_TM_SimpleName(
                  mk_TM_SimpleName(
                    getText
                    (
                  id))),
                TM_NoOptionalReferenceType)))
                    )⟩⟩⟩⟩),
        mk_TM_MethodDeclaration(
          ⟨TM_JAVA_PUBLIC⟩,
```

```
                                    mk_TM_SimpleName(″toString″),
                                    Make_TM_ReferenceType(
                                      mk_TM_ReferenceType
                                      (
                        Make_TM_SimpleName(
                            mk_TM_SimpleName(
                                ″String″)),
                        TM_NoOptionalReferenceType)), ⟨⟩,
                                    Make_TM_OptionalBlock(
                                      mk_TM_Block(
                                        ⟨TM_ReturnStatement(
                                            Make_TM_JavaValueLiteral(
                                              Make_TM_JavaValueLiteralString(
                                                mk_TM_JavaValueLiteralString(
                                                  getText(
                                                    id)))))
                                        ⟩)))⟩, ⟨⟩, ⟨⟩)))⟩
                  ⟩
          end
    end,

makeClassDeclaration2 :
    TM_Id × TM_OptionalPackageDeclaration ×
    TM_ImportDeclaration* ×
    TM_OptionalSimpleName × TM_OptionalId →
      TM_CompilationUnit
makeClassDeclaration2(id, opd, idl, ext, visitor) ≡
    case visitor of
      Make_TM_Id(visitorId) →
        mk_TM_CompilationUnit(
            opd, idl,
            ⟨TM_ClassDeclaration(
                    ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                    rslIdToSimpleName(id), ext, ⟨⟩, ⟨⟩,
                    ⟨mk_TM_MethodDeclaration(
                            ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                            mk_TM_SimpleName(″equals″),
                            Make_TM_PrimitiveType(TM_JAVA_BOOL),
                            ⟨mk_TM_SingleVariableDeclaration(
                                ⟨⟩,
                                Make_TM_ReferenceType(
```

```
                                    mk_TM_ReferenceType
                                    (
                        Make_TM_SimpleName(
                              mk_TM_SimpleName(
                                  "Object")),
                        TM_NoOptionalReferenceType)),
                                    mk_TM_SimpleName("o"),
                                    TM_NoOptionalExpression)⟩,
                              TM_NoOptionalBlock),
                           mk_TM_MethodDeclaration(
                              ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                              mk_TM_SimpleName("toString"),
                              Make_TM_ReferenceType(
                                mk_TM_ReferenceType(
                                  Make_TM_SimpleName(
                                    mk_TM_SimpleName("String")),
                                  TM_NoOptionalReferenceType)),
                              ⟨⟩, TM_NoOptionalBlock)⟩ ⌢
                           makeVisitorMethod(id, visitorId, true),
                           ⟨⟩, ⟨⟩)⟩),
     TM_NoOptionalId →
        mk_TM_CompilationUnit(
           opd, idl,
           ⟨TM_ClassDeclaration(
                    ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                    rslIdToSimpleName(id), ext, ⟨⟩, ⟨⟩,
                    ⟨mk_TM_MethodDeclaration(
                            ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                            mk_TM_SimpleName("equals"),
                            Make_TM_PrimitiveType(TM_JAVA_BOOL),
                            ⟨mk_TM_SingleVariableDeclaration(
                                ⟨⟩,
                                Make_TM_ReferenceType(
                                  mk_TM_ReferenceType
                                  (
                        Make_TM_SimpleName(
                              mk_TM_SimpleName(
                                  "Object")),
                        TM_NoOptionalReferenceType)),
                                    mk_TM_SimpleName("o"),
                                    TM_NoOptionalExpression)⟩,
```

```
                                TM_NoOptionalBlock),
                            mk_TM_MethodDeclaration(
                                ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
                                mk_TM_SimpleName(″toString″),
                                Make_TM_ReferenceType(
                                  mk_TM_ReferenceType(
                                    Make_TM_SimpleName(
                                      mk_TM_SimpleName(″String″)),
                                    TM_NoOptionalReferenceType)),
                                ⟨⟩, TM_NoOptionalBlock)⟩, ⟨⟩,
                        ⟨⟩)))
    end,

makeClassDeclarationList :
    TM_Variant* × TM_Id ×
    TM_OptionalPackageDeclaration ×
    TM_ImportDeclaration* ×
    TM_OptionalSimpleName × TM_OptionalId →
      TM_CompilationUnit*
makeClassDeclarationList(
    vl, id, opd, idl, ext, visitor) ≡
    if vl = ⟨⟩ then ⟨⟩
    else
      case hd vl of
        Make_TM_Constructor(c) →
          case visitor of
            Make_TM_Id(visitorId) →
              ⟨mk_TM_CompilationUnit(
                    opd, idl,
                    ⟨TM_ClassDeclaration(
                          ⟨TM_JAVA_PUBLIC⟩,
                          rslIdToSimpleName(id(c)), ext, ⟨
                          ⟩, ⟨⟩,
                          makeStandardMethodDeclarationListConstant(
                            id(c)) ⌢
                          makeVisitorMethod(
                            id(c), visitorId, false),
                          ⟨⟩, ⟨⟩)⟩)⟩ ⌢
                makeClassDeclarationList(
                    tl vl, id, opd, idl, ext, visitor),
              TM_NoOptionalId →
```

```
                ⟨mk_TM_CompilationUnit(
                        opd, idl,
                        ⟨TM_ClassDeclaration(
                                ⟨TM_JAVA_PUBLIC⟩,
                                rslIdToSimpleName(id(c)), ext, ⟨
                                ⟩, ⟨⟩,
                                makeStandardMethodDeclarationListConstant(
                                  id(c)), ⟨⟩, ⟨⟩)⟩)⟩ ⌢
                makeClassDeclarationList(
                    tl vl, id, opd, idl, ext, visitor)
            end,
        TM_RecordVariant(c, ckl) →
            case visitor of
                Make_TM_Id(visitorId) →
                    ⟨mk_TM_CompilationUnit(
                            opd, idl,
                            ⟨TM_ClassDeclaration(
                                    ⟨TM_JAVA_PUBLIC⟩,
                                    rslIdToSimpleName(id(c)), ext, ⟨
                                    ⟩,
                                    makeConstructorDeclaration(
                                      id(c), ckl),
                                    makeStandardMethodDeclarationList(
                                      id(c), ckl) ⌢
                                    makeVisitorMethod(
                                      id(c), visitorId, false),
                                    makeFieldDeclaration(ckl, 0), ⟨
                                    ⟩)⟩)⟩ ⌢
                    makeClassDeclarationList(
                        tl vl, id, opd, idl, ext, visitor),
                TM_NoOptionalId →
                    ⟨mk_TM_CompilationUnit(
                            opd, idl,
                            ⟨TM_ClassDeclaration(
                                    ⟨TM_JAVA_PUBLIC⟩,
                                    rslIdToSimpleName(id(c)), ext, ⟨
                                    ⟩,
                                    makeConstructorDeclaration(
                                      id(c), ckl),
                                    makeStandardMethodDeclarationList(
                                      id(c), ckl),
```

```
                                    makeFieldDeclaration(ckl, 0), ⟨
                                    ⟩)))⟩ ⌢
                            makeClassDeclarationList(
                                tl vl, id, opd, idl, ext, visitor)
                    end
                end
            end,

        makeStandardMethodDeclarationListConstant :
            TM_Id → TM_MethodDeclaration*
        makeStandardMethodDeclarationListConstant(id) ≡
            ⟨makeEqualsMethodConstant(id)⟩ ⌢
            ⟨makeToStringMethodConstant(id)⟩,

        makeEqualsMethodConstant :
            TM_Id → TM_MethodDeclaration
        makeEqualsMethodConstant(id) ≡
            mk_TM_MethodDeclaration(
                ⟨TM_JAVA_PUBLIC⟩, mk_TM_SimpleName("equals"),
                Make_TM_PrimitiveType(TM_JAVA_BOOL),
                ⟨mk_TM_SingleVariableDeclaration(
                        ⟨⟩,
                        Make_TM_ReferenceType(
                            mk_TM_ReferenceType(
                                Make_TM_SimpleName(
                                    mk_TM_SimpleName("Object")),
                                TM_NoOptionalReferenceType)),
                        mk_TM_SimpleName("o"),
                        TM_NoOptionalExpression)⟩,
                Make_TM_OptionalBlock(
                    mk_TM_Block(
                        ⟨TM_ReturnStatement(
                                TM_InstanceOfExpression(
                                    Make_TM_JavaName(
                                        Make_TM_SimpleName(
                                            mk_TM_SimpleName("o"))),
                                    Make_TM_ReferenceType(
                                        mk_TM_ReferenceType(
                                            Make_TM_SimpleName(
                                                mk_TM_SimpleName(
                                                    getText(id))),
```

```
                                        TM_NoOptionalReferenceType))))
                ⟩))),

makeToStringMethodConstant :
    TM_Id → TM_MethodDeclaration
makeToStringMethodConstant(id) ≡
    mk_TM_MethodDeclaration(
        ⟨TM_JAVA_PUBLIC⟩, mk_TM_SimpleName(″toString″),
        Make_TM_ReferenceType(
            mk_TM_ReferenceType(
                Make_TM_SimpleName(
                    mk_TM_SimpleName(″String″)),
                TM_NoOptionalReferenceType)), ⟨⟩,
        Make_TM_OptionalBlock(
            mk_TM_Block(
                ⟨TM_ReturnStatement(
                    Make_TM_JavaValueLiteral(
                        Make_TM_JavaValueLiteralString(
                            mk_TM_JavaValueLiteralString(
                                getText(id)))))))),

makeFieldDeclaration :
    TM_ComponentKind* × Int →
        TM_FieldDeclaration*
makeFieldDeclaration(ckl, i) ≡
    if ckl = ⟨⟩ then ⟨⟩
    else
        ⟨mk_TM_FieldDeclaration(
            ⟨TM_JAVA_PRIVATE⟩,
            matchType(type_expr(hd ckl), false, true),
            mk_TM_VariableDeclarationFragment(
                mk_TM_SimpleName(″_v″ ⌢ convert(i)),
                TM_NoOptionalExpression))⟩ ⌢
        makeFieldDeclaration(tl ckl, i + 1)
    end,

makeConstructorDeclaration :
    TM_Id × TM_ComponentKind* →
        TM_ConstructorDeclaration*
makeConstructorDeclaration(id, ckl) ≡
    ⟨mk_TM_ConstructorDeclaration(
```

```
        ⟨TM_JAVA_PUBLIC⟩, rslIdToSimpleName(id),
        makeArgumentList(ckl, 0),
        mk_TM_Block(
            makeConstructorStatementList(ckl, 0)))⟩,

makeConstructorStatementList :
  TM_ComponentKind* × Int → TM_Statement*
makeConstructorStatementList(ckl, i) ≡
  if ckl = ⟨⟩ then ⟨⟩
  else
    ⟨TM_ExpressionStatement(
        TM_AssignmentExpression(
            TM_FieldAccessExpression(
                Make_TM_OptionalExpression(
                    TM_ThisExpression(
                        TM_NoOptionalSimpleName)),
                mk_TM_SimpleName("_v" ⌢ convert(i))),
            TM_JAVA_ASSIGNMENT_OP_EQUAL,
            Make_TM_JavaName(
                Make_TM_SimpleName(
                    mk_TM_SimpleName("_v" ⌢ convert(i))))
        ))⟩ ⌢
    makeConstructorStatementList(tl ckl, i + 1)
  end,

makeStandardMethodDeclarationList :
  TM_Id × TM_ComponentKind* →
    TM_MethodDeclaration*
makeStandardMethodDeclarationList(id, ckl) ≡
  ⟨mk_TM_MethodDeclaration(
        ⟨TM_JAVA_PUBLIC⟩, mk_TM_SimpleName("equals"),
        Make_TM_PrimitiveType(TM_JAVA_BOOL),
        ⟨mk_TM_SingleVariableDeclaration(
            ⟨⟩,
            Make_TM_ReferenceType(
                mk_TM_ReferenceType(
                    Make_TM_SimpleName(
                        mk_TM_SimpleName("Object")),
                    TM_NoOptionalReferenceType)),
            mk_TM_SimpleName("o"),
            TM_NoOptionalExpression)⟩,
```

```
            Make_TM_OptionalBlock(
                mk_TM_Block(
                    makeStatementListEqualsMethod(id, ckl) ⌢
                    ⟨TM_ReturnStatement(
                            Make_TM_JavaValueLiteral(
                              Make_TM_JavaValueLiteralBool(
                                mk_TM_JavaValueLiteralBool(
                                  "true")))))))),
        mk_TM_MethodDeclaration(
            ⟨TM_JAVA_PUBLIC⟩,
            mk_TM_SimpleName("toString"),
            Make_TM_ReferenceType(
                mk_TM_ReferenceType(
                    Make_TM_SimpleName(
                        mk_TM_SimpleName("String")),
                    TM_NoOptionalReferenceType)), ⟨⟩,
            Make_TM_OptionalBlock(
                mk_TM_Block(
                    ⟨TM_ReturnStatement(
                            TM_InfixExpression(
                              Make_TM_JavaValueLiteral(
                                Make_TM_JavaValueLiteralString(
                                  mk_TM_JavaValueLiteralString(
                                    getText(id) ⌢ "(")),
                              TM_JAVA_PLUS,
                              makeExpressionToStringMethod(ckl))
                        ))))⟩ ⌢
    makeAccessorMethodDeclarationList(ckl, 0),

makeStatementListEqualsMethod :
    TM_Id × TM_ComponentKind* → TM_Statement*
makeStatementListEqualsMethod(id, ckl) ≡
    if ckl = ⟨⟩ then ⟨⟩
    else
        case matchType(type_expr(hd ckl), false, false) of
            Make_TM_PrimitiveType(t) →
                ⟨TM_IfStatement(
                        TM_PrefixExpression(
                            TM_JAVA_NOT,
                            TM_ParenthesizedExpression(
                                TM_InfixExpression(
```

```
                    TM_MethodInvocation(
                      Make_TM_OptionalExpression(
                        TM_ThisExpression(
                          TM_NoOptionalSimpleName)
                      ),
                      makeSimpleNameFromDestructor(
                        optional_destructor(hd ckl)
                      ), ⟨⟩), TM_JAVA_EQUALS,
                    TM_MethodInvocation(
                      Make_TM_OptionalExpression(
                        TM_ParenthesizedExpression(
                          TM_CastExpression(
                            Make_TM_ReferenceType(
                              mk_TM_ReferenceType
                              (
                    Make_TM_SimpleName(
                      rslIdToSimpleName(
                        id)),
                    TM_NoOptionalReferenceType)),
                              Make_TM_JavaName(
                                Make_TM_SimpleName(
                                  mk_TM_SimpleName(
                                    "o"))))))),
                        makeSimpleNameFromDestructor(
                          optional_destructor(hd ckl)
                        ), ⟨⟩)))),
          mk_TM_Block(
              ⟨TM_ReturnStatement(
                  Make_TM_JavaValueLiteral(
                    Make_TM_JavaValueLiteralBool(
                      mk_TM_JavaValueLiteralBool(
                        "false"))))⟩),
          TM_NoOptionalBlock)⟩ ⌢
    makeStatementListEqualsMethod(id, tl ckl),
Make_TM_ReferenceType(t) →
    ⟨TM_IfStatement(
          TM_MethodInvocation(
              Make_TM_OptionalExpression(
                TM_MethodInvocation(
                  Make_TM_OptionalExpression(
                    TM_ThisExpression(
```

```
                                        TM_NoOptionalSimpleName)),
                                    makeSimpleNameFromDestructor(
                                      optional_destructor(hd ckl)),
                                    ⟨⟩)),
                                mk_TM_SimpleName("equals"),
                                ⟨TM_MethodInvocation(
                                    Make_TM_OptionalExpression(
                                      TM_ParenthesizedExpression(
                                        TM_CastExpression(
                                          Make_TM_ReferenceType(
                                            mk_TM_ReferenceType
                                            (
                                  Make_TM_SimpleName(
                                    rslIdToSimpleName(
                                      id)),
                                  TM_NoOptionalReferenceType)),
                                            Make_TM_JavaName(
                                              Make_TM_SimpleName(
                                                mk_TM_SimpleName(
                                                  "o")))))),
                                    makeSimpleNameFromDestructor(
                                      optional_destructor(hd ckl)),
                                    ⟨⟩)⟩),
                            mk_TM_Block(
                                ⟨TM_ReturnStatement(
                                    Make_TM_JavaValueLiteral(
                                      Make_TM_JavaValueLiteralBool(
                                        mk_TM_JavaValueLiteralBool(
                                          "false"))))⟩),
                            TM_NoOptionalBlock)⟩ ⌢
                    makeStatementListEqualsMethod(id, tl ckl)
        end
    end,

makeVisitorMethod :
    TM_Id × TM_Id × Bool → TM_MethodDeclaration*
makeVisitorMethod(id, visitor, abstractMethod) ≡
    if abstractMethod
    then
        ⟨mk_TM_MethodDeclaration(
                ⟨TM_JAVA_PUBLIC, TM_JAVA_ABSTRACT⟩,
```

```
            mk_TM_SimpleName(″accept″),
            Make_TM_PrimitiveType(TM_JAVA_VOID),
            ⟨mk_TM_SingleVariableDeclaration(
                ⟨⟩,
                Make_TM_ReferenceType(
                    mk_TM_ReferenceType(
                        Make_TM_SimpleName(
                            rslIdToSimpleName(visitor)),
                        TM_NoOptionalReferenceType)),
                mk_TM_SimpleName(″visitor″),
                TM_NoOptionalExpression)⟩,
            TM_NoOptionalBlock)⟩
else
    ⟨mk_TM_MethodDeclaration(
            ⟨TM_JAVA_PUBLIC⟩,
            mk_TM_SimpleName(″accept″),
            Make_TM_PrimitiveType(TM_JAVA_VOID),
            ⟨mk_TM_SingleVariableDeclaration(
                ⟨⟩,
                Make_TM_ReferenceType(
                    mk_TM_ReferenceType(
                        Make_TM_SimpleName(
                            rslIdToSimpleName(visitor)),
                        TM_NoOptionalReferenceType)),
                mk_TM_SimpleName(″visitor″),
                TM_NoOptionalExpression)⟩,
            Make_TM_OptionalBlock(
                mk_TM_Block(
                    ⟨TM_ExpressionStatement(
                        TM_MethodInvocation(
                            Make_TM_OptionalExpression(
                                Make_TM_JavaName(
                                    Make_TM_SimpleName(
                                        mk_TM_SimpleName(
                                            ″visitor″)))),
                            mk_TM_SimpleName(
                                ″visit″ ^ getText(id)),
                            ⟨TM_ThisExpression(
                                TM_NoOptionalSimpleName)⟩)
                    ))))⟩
end,
```

```
makeSimpleNameFromDestructor :
   TM_OptionalDestructor → TM_SimpleName
makeSimpleNameFromDestructor(od) ≡
   case od of
      TM_Destructor(d) → rslIdToSimpleName(d),
      TM_NoDestructor →
         mk_TM_SimpleName("UNKNOWN_METHOD_NAME")
   end,

makeExpressionToStringMethod :
   TM_ComponentKind* → TM_Expression
makeExpressionToStringMethod(ckl) ≡
   if ckl = ⟨⟩
   then
      Make_TM_JavaValueLiteral(
          Make_TM_JavaValueLiteralString(
              mk_TM_JavaValueLiteralString(")")))
   else
      case matchType(type_expr(hd ckl), false, false) of
         Make_TM_PrimitiveType(pt) →
            if tl ckl = ⟨⟩
            then
               TM_InfixExpression(
                  TM_MethodInvocation(
                        Make_TM_OptionalExpression(
                           TM_ThisExpression(
                             TM_NoOptionalSimpleName)),
                        makeSimpleNameFromDestructor(
                           optional_destructor(hd ckl)), ⟨⟩),
                     TM_JAVA_PLUS,
                     makeExpressionToStringMethod(tl ckl))
            else
               TM_InfixExpression(
                  TM_MethodInvocation(
                        Make_TM_OptionalExpression(
                           TM_ThisExpression(
                             TM_NoOptionalSimpleName)),
                        makeSimpleNameFromDestructor(
                           optional_destructor(hd ckl)), ⟨⟩),
                     TM_JAVA_PLUS,
```

```
            TM_InfixExpression(
                Make_TM_JavaValueLiteral(
                    Make_TM_JavaValueLiteralString(
                      mk_TM_JavaValueLiteralString(
                        ″, ″))), TM_JAVA_PLUS,
                makeExpressionToStringMethod(tl ckl)))
    end,
Make_TM_ReferenceType(rt) →
    if tl ckl = ⟨⟩
    then
        TM_InfixExpression(
            TM_MethodInvocation(
                Make_TM_OptionalExpression(
                    TM_MethodInvocation(
                      Make_TM_OptionalExpression(
                        TM_ThisExpression(
                          TM_NoOptionalSimpleName)),
                      makeSimpleNameFromDestructor(
                        optional_destructor(hd ckl)),
                      ⟨⟩)),
                mk_TM_SimpleName(″toString″), ⟨⟩),
            TM_JAVA_PLUS,
            makeExpressionToStringMethod(tl ckl))
    else
        TM_InfixExpression(
            TM_MethodInvocation(
                Make_TM_OptionalExpression(
                    TM_MethodInvocation(
                      Make_TM_OptionalExpression(
                        TM_ThisExpression(
                          TM_NoOptionalSimpleName)),
                      makeSimpleNameFromDestructor(
                        optional_destructor(hd ckl)),
                      ⟨⟩)),
                mk_TM_SimpleName(″toString″), ⟨⟩),
            TM_JAVA_PLUS,
            TM_InfixExpression(
                Make_TM_JavaValueLiteral(
                    Make_TM_JavaValueLiteralString(
                      mk_TM_JavaValueLiteralString(
                        ″, ″))), TM_JAVA_PLUS,
```

```
                                    makeExpressionToStringMethod(tl ckl)))
                end
            end
        end,

    makeAccessorMethodDeclarationList :
        TM_ComponentKind* × Int →
            TM_MethodDeclaration*
    makeAccessorMethodDeclarationList(ckl, i) ≡
        if ckl = ⟨⟩ then ⟨⟩
        else
            ⟨mk_TM_MethodDeclaration(
                    ⟨TM_JAVA_PUBLIC⟩,
                    makeSimpleNameFromDestructor(
                            optional_destructor(hd ckl)),
                    matchType(type_expr(hd ckl), false, true), ⟨
                    ⟩,
                    Make_TM_OptionalBlock(
                            mk_TM_Block(
                                ⟨TM_ReturnStatement(
                                        Make_TM_JavaName(
                                            Make_TM_SimpleName(
                                                mk_TM_SimpleName(
                                                    "_v" ⌢ convert(i)))))⟩))))
            ⟩ ⌢
            makeAccessorMethodDeclarationList(tl ckl, i + 1)
        end,

    matchTypeInner :
        TM_TypeEvaluator × Bool × Bool → TM_JavaType
    matchTypeInner(te, useWrapper, useInterface) ≡
        case te of
            TM_TypeEvaluator_TM_TypeExpr(inner_type_expr) →
                case inner_type_expr of
                    TM_TypeExprList(tel) →
                        case tel of
                            TM_FiniteListTypeExpr(flte) →
                                matchType(flte, useWrapper, useInterface),
                            _ →
                                Make_TM_ReferenceType(
                                    mk_TM_ReferenceType(
```

```
                                Make_TM_SimpleName(
                                  mk_TM_SimpleName
                                  (
                  "UNKNOWN_TYPE_MATCH_TYPE_INNER_1")),
                                  TM_NoOptionalReferenceType))
                  end,
              _ →
                  Make_TM_ReferenceType(
                      mk_TM_ReferenceType(
                          Make_TM_SimpleName(
                              mk_TM_SimpleName(
                                "UNKNOWN_TYPE_MATCH_TYPE_INNER_2"
                              )), TM_NoOptionalReferenceType))
          end,
      _ →
          Make_TM_ReferenceType(
              mk_TM_ReferenceType(
                  Make_TM_SimpleName(
                      mk_TM_SimpleName(
                          "UNKNOWN_TYPE_MATCH_TYPE_INNER_3")),
                  TM_NoOptionalReferenceType))
  end,

matchType :
  TM_TypeEvaluator × Bool × Bool → TM_JavaType
matchType(te, useWrapper, useInterface) ≡
  case te of
    TM_TypeEvaluator_TM_TypeExpr(type_expr) →
      matchType(type_expr, useWrapper, useInterface),
    TM_TypeEvaluator_TM_Constructor(c) →
      Make_TM_ReferenceType(
          mk_TM_ReferenceType(
              Make_TM_SimpleName(
                  rslIdToSimpleName(id(c))),
              TM_NoOptionalReferenceType))
  end,

matchType : TM_TypeExpr × Bool × Bool → TM_JavaType
matchType(te, useWrapper, useInterface) ≡
  case te of
    TM_TypeLiteral(literal) →
```

```
case literal of
  TM_RSL_UNIT →
    Make_TM_PrimitiveType(TM_JAVA_VOID),
  TM_RSL_INT →
    if useWrapper
    then
      Make_TM_ReferenceType(
        mk_TM_ReferenceType(
          Make_TM_SimpleName(
            mk_TM_SimpleName("Integer")),
          TM_NoOptionalReferenceType))
    else Make_TM_PrimitiveType(TM_JAVA_INT)
    end,
  TM_RSL_REAL →
    if useWrapper
    then
      Make_TM_ReferenceType(
        mk_TM_ReferenceType(
          Make_TM_SimpleName(
            mk_TM_SimpleName("Double")),
          TM_NoOptionalReferenceType))
    else Make_TM_PrimitiveType(TM_JAVA_DOUBLE)
    end,
  TM_RSL_CHAR →
    if useWrapper
    then
      Make_TM_ReferenceType(
        mk_TM_ReferenceType(
          Make_TM_SimpleName(
            mk_TM_SimpleName("Character")),
          TM_NoOptionalReferenceType))
    else Make_TM_PrimitiveType(TM_JAVA_CHAR)
    end,
  TM_RSL_TEXT →
    Make_TM_ReferenceType(
      mk_TM_ReferenceType(
        Make_TM_SimpleName(
          mk_TM_SimpleName("String")),
        TM_NoOptionalReferenceType)),
  TM_RSL_BOOL →
    if useWrapper
```

```
        then
          Make_TM_ReferenceType(
            mk_TM_ReferenceType(
              Make_TM_SimpleName(
                mk_TM_SimpleName("Boolean")),
              TM_NoOptionalReferenceType))
        else Make_TM_PrimitiveType(TM_JAVA_BOOL)
        end
    end,
  TM_TypeExprList(tel) →
    case tel of
      TM_FiniteListTypeExpr(flte) →
        if useInterface
        then
          Make_TM_ReferenceType(
            mk_TM_ReferenceType(
              Make_TM_SimpleName(
                mk_TM_SimpleName("RSLList")),
              Make_TM_OptionalReferenceType(
                matchTypeReferenceType(
                  flte, true, useInterface))))
        else
          Make_TM_ReferenceType(
            mk_TM_ReferenceType(
              Make_TM_SimpleName(
                mk_TM_SimpleName("RSLListDefault")
              ),
              Make_TM_OptionalReferenceType(
                matchTypeReferenceType(
                  flte, true, useInterface))))
        end
    end,
  TM_TypeName(id) →
    Make_TM_ReferenceType(
      mk_TM_ReferenceType(
        Make_TM_SimpleName(rslIdToSimpleName(id)),
        TM_NoOptionalReferenceType)),
  _ →
    Make_TM_ReferenceType(
      mk_TM_ReferenceType(
        Make_TM_SimpleName(
```

```
                    mk_TM_SimpleName(
                        "UNKNOWN_TYPE_MATCH_TYPE")),
                TM_NoOptionalReferenceType))
  end,

makeName : Text* → TM_JavaName
makeName(l) ≡
  if l = ⟨⟩
  then
    Make_TM_SimpleName(mk_TM_SimpleName("nopackage"))
  else
    if tl l = ⟨⟩
    then Make_TM_SimpleName(mk_TM_SimpleName(hd l))
    else
      Make_TM_QualifiedName(
        mk_TM_QualifiedName(
          makeName(tl l), mk_TM_SimpleName(hd l)))
    end
  end,

matchTypeReferenceType :
  TM_TypeEvaluator × Bool × Bool → TM_ReferenceType
matchTypeReferenceType(te, useWrapper, useInterface) ≡
  case te of
    TM_TypeEvaluator_TM_TypeExpr(t) →
      matchTypeReferenceType(
        t, useWrapper, useInterface),
    TM_TypeEvaluator_TM_Constructor(c) →
      mk_TM_ReferenceType(
        Make_TM_SimpleName(rslIdToSimpleName(id(c))),
        TM_NoOptionalReferenceType),
    _ →
      mk_TM_ReferenceType(
        Make_TM_SimpleName(
          mk_TM_SimpleName(
            "UNKNOWN_TYPE_MATCH_TYPE_REFERENCE")),
        TM_NoOptionalReferenceType)
  end,

matchTypeReferenceType :
  TM_TypeExpr × Bool × Bool → TM_ReferenceType
```

```
matchTypeReferenceType(te, useWrapper, useInterface) ≡
  case matchType(te, useWrapper, useInterface) of
    Make_TM_ReferenceType(rt) → rt
  end,

rslIdToSimpleName : TM_Id → TM_SimpleName
rslIdToSimpleName(id) ≡ mk_TM_SimpleName(getText(id)),

rslIdToSimpleName : TM_Pattern → TM_SimpleName
rslIdToSimpleName(p) ≡
  case p of
    TM_NamePattern(id, vi) → rslIdToSimpleName(id),
    _ →
      mk_TM_SimpleName(
        "UNKNOWN_VARIABLE_NAME_IN_PATTERN")
  end,

valueInitializerToSimpleName :
  TM_Pattern → TM_SimpleName
valueInitializerToSimpleName(p) ≡
  case p of
    TM_NamePattern(id, vi) →
      case vi of
        Make_TM_Id(id2) → rslIdToSimpleName(id2),
        _ →
          mk_TM_SimpleName(
            "UNKNOWN_VARIABLE_INITIALIZER")
      end,
    _ →
      mk_TM_SimpleName("UNKNOWN_VARIABLE_INITIALIZER")
  end,

textInList : Text × Text* → Bool
textInList(t, tList) ≡
  if tList = ⟨⟩ then false
  else
    if hd tList = t then true
    else textInList(t, tl tList)
    end
  end,
```

```
        convert : Int → Text
        convert(i) ≡
            case i of
                0 → "0",
                1 → "1",
                2 → "2",
                3 → "3",
                4 → "4",
                5 → "5",
                6 → "6",
                7 → "7",
                8 → "8",
                9 → "9",
                _ → "TOO_MANY_LEVELS"
            end
    end
```

C.2.2 RSLAst_Module2

```
scheme RSLAst_Module2 =
    class
        type
            TM_RSLAst :: libmodule : TM_LibModule,
            TM_LibModule ::
                context_list : TM_Id*   schemedef : TM_SchemeDef,
            TM_SchemeDef :: id : TM_Id   class_expr : TM_ClassExpr,
            TM_ClassExpr ==
                TM_BasicClassExpr(declaration_list : TM_Decl*) |
                TM_ExtendingClassExpr(
                    base_class : TM_ClassExpr,
                    extension_class : TM_ClassExpr) |
                TM_SchemeInstantiation(id : TM_Id),
            TM_Decl ==
                TM_ValueDecl(value_def_list : TM_ValueDef*) |
                TM_TypeDecl(type_def_list : TM_TypeDef*) |
                TM_TestDecl(test_def_list : TM_TestDef*),
            /*Type Definitions*/
            TM_TypeDef ==
                TM_SortDef(sd_id : TM_Id) |
                TM_VariantDef(
```

```
        id : TM_Id, variant_list : TM_Variant*) |
    TM_ShortRecordDef(
        srd_id : TM_Id,
        component_kind_string : TM_ComponentKind*),
TM_Variant ==
    Make_TM_Constructor(constructor : TM_Constructor) |
    TM_RecordVariant(
        record_constructor : TM_Constructor,
        component_kind_list : TM_ComponentKind*),
TM_ComponentKind ::
    optional_destructor : TM_OptionalDestructor
    type_expr : TM_TypeExpr
    optional_reconstructor : TM_OptionalReconstructor,
TM_Constructor :: id : TM_Id,
TM_OptionalDestructor ==
    TM_Destructor(id : TM_Id) | TM_NoDestructor,
TM_OptionalReconstructor ==
    TM_Reconstructor(id : TM_Id) | TM_NoReconstructor,
/*Value Definitions*/
TM_ValueDef ==
    TM_ExplicitFunctionDef(
        single_typing : TM_SingleTyping,
        formal_function_application :
            TM_FormalFunctionApplication,
        value_expr : TM_ValueExpr),
TM_SingleTyping ::
    binding : TM_Binding   type_expr : TM_TypeExpr,
TM_FormalFunctionApplication ==
    TM_IdApplication(
        id : TM_Id,
        formal_function_parameter_list :
            TM_FormalFunctionParameter*),
TM_FormalFunctionParameter ::
    binding_list : TM_Binding*,
TM_Binding :: id : TM_Id,
/*Test Definitions*/
TM_TestDef ==
    TM_TestCase(id : TM_Id, value_expr : TM_ValueExpr),
/*Type Expressions*/
TM_TypeExpr ==
    TM_TypeExprList(type_expr_list : TM_TypeExprLists) |
```

```
    TM_FunctionTypeExpr(
        type_expr_argument : TM_TypeExpr,
        function_arrow : TM_FunctionArrow,
        type_expr_result : TM_TypeExpr) |
    TM_TypeLiteral(type_literal : TM_TypeLiterals) |
    TM_TypeExprProduct(component_list : TM_TypeExpr*) |
    TM_TypeName(id : TM_Id),
TM_FunctionArrow == TM_TOTAL_FUNCTION_ARROW,
TM_TypeExprLists ==
    TM_FiniteListTypeExpr(type_expr : TM_TypeExpr),
TM_TypeLiterals ==
    TM_RSL_UNIT |
    TM_RSL_INT |
    TM_RSL_NAT |
    TM_RSL_REAL |
    TM_RSL_BOOL |
    TM_RSL_CHAR |
    TM_RSL_TEXT,
/*Value Expression*/
TM_ValueExpr ==
    Make_TM_IfExpr(if_expr : TM_IfExpr) |
    Make_TM_ValueLiteral(value_literal : TM_ValueLiteral) |
    TM_ValueInfixExpr(
        left : TM_ValueExpr,
        op : TM_InfixOperator,
        right : TM_ValueExpr) |
    TM_ValuePrefixExpr(
        op : TM_PrefixOperator, operand : TM_ValueExpr) |
    Make_TM_ListExpr(list_expr : TM_ListExpr) |
    TM_ApplicationExpr(
        value_expr : TM_ValueExpr,
        value_expr_list : TM_ValueExpr*) |
    Make_TM_ValueOrVariableName(
        value_or_variable_name : TM_ValueOrVariableName) |
    TM_ParenthesizedExpr(
        parenthesized_expr : TM_ValueExpr) |
    TM_CaseExpr(
        condition : TM_ValueExpr,
        case_branch_list : TM_CaseBranch*),
TM_ValueLiteral ==
    TM_ValueLiteralInteger(getTextInteger : Text) |
```

```
        TM_ValueLiteralReal(getTextReal : Text) |
        TM_ValueLiteralBool(getTextBool : Text) |
        TM_ValueLiteralChar(getTextChar : Text) |
        TM_ValueLiteralText(getTextText : Text),
    TM_ValueOrVariableName :: id : TM_Id,
    TM_IfExpr ::
        condition : TM_ValueExpr
        if_case : TM_ValueExpr
        elsif_list : TM_Elsif*
        else_case : TM_ValueExpr,
    TM_Elsif ::
        condition : TM_ValueExpr   elsif_case : TM_ValueExpr,
    TM_CaseBranch ::
        pattern : TM_Pattern   value_expr : TM_ValueExpr,
    TM_Pattern ==
        TM_ValueLiteralPattern(
            value_literal : TM_ValueLiteral) |
        TM_NamePattern(
            id : TM_Id, value_initializer : TM_OptionalId) |
        TM_RecordPattern(
            value_or_variable_name : TM_ValueOrVariableName,
            inner_pattern_list : TM_Pattern*) |
        TM_WildcardPattern,
    TM_ListExpr ==
        TM_EnumeratedListExpr(
            value_expr_list : TM_ValueExpr*),
    TM_InfixOperator ==
        TM_RSL_PLUS |
        TM_RSL_EQUAL |
        TM_RSL_HAT |
        TM_RSL_STAR |
        TM_RSL_SLASH,
    TM_PrefixOperator == TM_RSL_HD | TM_RSL_TL,
    /*Commom*/
    TM_OptionalId ==
        Make_TM_Id(id : TM_Id) | TM_NoOptionalId,
    TM_Id :: getText : Text
end
```

C.2.3 RSLAst_WrapperModule2

context: RSLAst_Module2

```
scheme RSLAst_WrapperModule2 =
  extend RSLAst_Module2 with
  class
    type
      /∗Type evaluation∗/
      TM_TypeEvaluator ==
        TM_TypeEvaluator_TM_TypeExpr(type_expr : TM_TypeExpr) |
        TM_TypeEvaluator_TM_Constructor(
            constructor : TM_Constructor) |
        TM_TypeEvaluator_TM_Destructor(
            destructor : TM_OptionalDestructor)
  end
```

C.2.4 RSLAst_WrapperModule_22

context: RSLAst_WrapperModule2, JavaAst_Module2

```
scheme RSLAst_WrapperModule_22 =
  extend JavaAst_Module2 with
  extend RSLAst_WrapperModule2 with
  class
    value
      /∗
      getTypeEvaluatorForTypeEvaluation : TM_ValueExpr
      → TM_TypeExpr
      getTypeEvaluatorForTypeEvaluation(ve) is
      TM_TypeExprList(TM_FiniteListTypeExpr(TM_TypeLiteral(
      TM_RSL_INT))),
      getTypeEvaluatorForTypeEvaluation : TM_ListExpr
      → TM_TypeExpr
      getTypeEvaluatorForTypeEvaluation(ve) is
      TM_TypeExprList(TM_FiniteListTypeExpr(TM_TypeLiteral(
      TM_RSL_INT))),
      getTypeEvaluatorForTypeEvaluation : TM_ValueOrVariableName
      → TM_TypeExpr
      getTypeEvaluatorForTypeEvaluation(vovn) is
```

```
TM_TypeExprList(TM_FiniteListTypeExpr(TM_TypeLiteral(
TM_RSL_INT)))
*/
getTypeEvaluatorForTypeEvaluation :
    TM_ValueExpr → TM_TypeEvaluator
getTypeEvaluatorForTypeEvaluation(ve) ≡
    TM_TypeEvaluator_TM_TypeExpr(
        TM_TypeExprList(
            TM_FiniteListTypeExpr(
                TM_TypeLiteral(TM_RSL_INT)))),

getTypeEvaluatorForTypeEvaluation :
    TM_ListExpr → TM_TypeEvaluator
getTypeEvaluatorForTypeEvaluation(ve) ≡
    TM_TypeEvaluator_TM_TypeExpr(
        TM_TypeExprList(
            TM_FiniteListTypeExpr(
                TM_TypeLiteral(TM_RSL_INT)))),

getTypeEvaluatorForTypeEvaluation :
    TM_ValueOrVariableName → TM_TypeEvaluator
getTypeEvaluatorForTypeEvaluation(vovn) ≡
    TM_TypeEvaluator_TM_TypeExpr(
        TM_TypeExprList(
            TM_FiniteListTypeExpr(
                TM_TypeLiteral(TM_RSL_INT)))),

getTypeEvaluatorForTypeEvaluation :
    TM_Pattern → TM_TypeEvaluator
getTypeEvaluatorForTypeEvaluation(p) ≡
    TM_TypeEvaluator_TM_TypeExpr(
        TM_TypeExprList(
            TM_FiniteListTypeExpr(
                TM_TypeLiteral(TM_RSL_INT)))),

Runner_instance_typeDecorate :
    TM_RSLAst → TM_CompilationUnit*
Runner_instance_typeDecorate(ra) ≡ ⟨⟩,

Runner_instance_translate :
    Text × Bool → TM_CompilationUnit*
```

```
        Runner_instance_translate(id, b) ≡ ⟨⟩
    end
```

C.2.5 JavaAst_Module2

```
scheme JavaAst_Module2 =
  class
    type
      TM_JavaAst ::
        compilationUnitList : TM_CompilationUnit*,
      TM_CompilationUnit ::
        optionalPackageDeclaration :
          TM_OptionalPackageDeclaration
        importDeclarationList : TM_ImportDeclaration*
        typeDeclarationList : TM_TypeDeclaration*,
      TM_OptionalPackageDeclaration ==
        TM_NoPackageDeclaration |
        TM_PackageDeclaration(name : TM_JavaName),
      TM_ImportDeclaration :: name : TM_JavaName,
      TM_TypeDeclaration ==
        TM_ClassDeclaration(
          modifierList : TM_Modifier*,
          name : TM_SimpleName,
          extendName : TM_OptionalSimpleName,
          implementList : TM_SimpleName*,
          constructorDeclarationList :
            TM_ConstructorDeclaration*,
          methodDeclarationList : TM_MethodDeclaration*,
          fieldDeclarationList : TM_FieldDeclaration*,
          typeDeclarationList : TM_TypeDeclaration*),
      TM_FieldDeclaration ::
        modifierList : TM_Modifier*
        getType : TM_JavaType
        variableDeclarationFragment :
          TM_VariableDeclarationFragment,
      TM_ConstructorDeclaration ::
        modifierList : TM_Modifier*
        name : TM_SimpleName
        argumentList : TM_SingleVariableDeclaration*
        block : TM_Block,
```

```
TM_MethodDeclaration ::
    modifierList : TM_Modifier*
    name : TM_SimpleName
    returnType : TM_JavaType
    argumentList : TM_SingleVariableDeclaration*
    block : TM_OptionalBlock,
TM_JavaType ==
    Make_TM_PrimitiveType(
        primitiveType : TM_PrimitiveType) |
    Make_TM_ReferenceType(
        referenceType : TM_ReferenceType) |
    Make_TM_ArrayType(arrayType : TM_ArrayType),
TM_PrimitiveType ==
    TM_JAVA_INT |
    TM_JAVA_VOID |
    TM_JAVA_DOUBLE |
    TM_JAVA_BOOL |
    TM_JAVA_CHAR,
TM_ReferenceType ::
    name : TM_JavaName
    optionalTypeArgument : TM_OptionalReferenceType,
TM_ArrayType :: getType : TM_JavaType,
TM_OptionalReferenceType ==
    Make_TM_OptionalReferenceType(
        referenceType : TM_ReferenceType) |
    TM_NoOptionalReferenceType,
TM_SingleVariableDeclaration ::
    modifierList : TM_Modifier*
    getType : TM_JavaType
    name : TM_SimpleName
    optionalInitialization : TM_OptionalExpression,
TM_OptionalBlock ==
    Make_TM_OptionalBlock(block : TM_Block) |
    TM_NoOptionalBlock,
TM_Block :: statementList : TM_Statement*,
TM_OptionalStatement ==
    Make_TM_Statement(statement : TM_Statement) |
    TM_NoOptionalStatement,
TM_Statement ==
    TM_ExpressionStatement(expression : TM_Expression) |
    TM_IfStatement(
```

```
        condition : TM_Expression,
        ifBlock : TM_Block,
        elseBlock : TM_OptionalBlock) |
    TM_ReturnStatement(
        expressionReturnStatement : TM_Expression) |
    Make_TM_VariableDeclarationStatement(
        variableDeclarationStatement :
            TM_VariableDeclarationStatement),
TM_VariableDeclarationStatement ::
    modifierList : TM_Modifier*
    getType : TM_JavaType
    fragment : TM_VariableDeclarationFragment,
TM_VariableDeclarationFragment ::
    name : TM_SimpleName
    optionalExpression : TM_OptionalExpression,
TM_OptionalExpression ==
    Make_TM_OptionalExpression(
        expression : TM_Expression) |
    TM_NoOptionalExpression,
TM_Expression ==
    Make_TM_JavaValueLiteral(
        valueLiteral : TM_JavaValueLiteral) |
    Make_TM_JavaName(name : TM_JavaName) |
    TM_ArrayCreation(
        getArrayType : TM_JavaType,
        getCount : TM_OptionalExpression,
        elementList : TM_Expression*) |
    TM_InfixExpression(
        left : TM_Expression,
        op : TM_JavaInfixOperator,
        right : TM_Expression) |
    TM_PrefixExpression(
        prefixOperator : TM_JavaPrefixOperator,
        prefixExpression : TM_Expression) |
    TM_MethodInvocation(
        optionalExpression : TM_OptionalExpression,
        name : TM_SimpleName,
        argumentList : TM_Expression*) |
    TM_ClassInstanceCreation(
        optionalExpressionClassInstanceCreation :
            TM_OptionalExpression,
```

```
        getType : TM_ReferenceType,
        argumentListClassInstanceCreation :
          TM_Expression*) |
    TM_AssignmentExpression(
        lhs : TM_Expression,
        assignmentOp : TM_AssignmentOperator,
        rhs : TM_Expression) |
    TM_ParenthesizedExpression(
        expression : TM_Expression) |
    TM_InstanceOfExpression(
        instanceOfExpression : TM_Expression,
        instanceOfType : TM_JavaType) |
    TM_ThisExpression(thisName : TM_OptionalSimpleName) |
    TM_CastExpression(
        castType : TM_JavaType,
        castExpression : TM_Expression) |
    TM_FieldAccessExpression(
        fieldExpression : TM_OptionalExpression,
        fieldName : TM_SimpleName),
TM_JavaValueLiteral ==
    Make_TM_JavaValueLiteralInteger(
        valueLiteralInteger : TM_JavaValueLiteralInteger) |
    Make_TM_JavaValueLiteralString(
        valueLiteralString : TM_JavaValueLiteralString) |
    Make_TM_JavaValueLiteralDouble(
        valueLiteralDouble : TM_JavaValueLiteralDouble) |
    Make_TM_JavaValueLiteralChar(
        valueLiteralChar : TM_JavaValueLiteralChar) |
    Make_TM_JavaValueLiteralBool(
        valueLiteralBool : TM_JavaValueLiteralBool) |
    TM_NullLiteral,
TM_JavaValueLiteralInteger :: getText : Text,
TM_JavaValueLiteralString :: getText : Text,
TM_JavaValueLiteralChar :: getText : Text,
TM_JavaValueLiteralDouble :: getText : Text,
TM_JavaValueLiteralBool :: getText : Text,
TM_JavaInfixOperator ==
    TM_JAVA_EQUALS |
    TM_JAVA_PLUS |
    TM_JAVA_STAR |
    TM_JAVA_DIV,
```

```
    TM_JavaPrefixOperator == TM_JAVA_NOT,
    TM_AssignmentOperator == TM_JAVA_ASSIGNMENT_OP_EQUAL,
    TM_JavaName ==
        Make_TM_SimpleName(name : TM_SimpleName) |
        Make_TM_QualifiedName(name : TM_QualifiedName),
    TM_SimpleName :: text : Text,
    TM_QualifiedName ::
        left : TM_JavaName   right : TM_SimpleName,
    TM_OptionalSimpleName ==
        Make_TM_OptionalSimpleName(name : TM_SimpleName) |
        TM_NoOptionalSimpleName,
    TM_Modifier ==
        TM_JAVA_PUBLIC |
        TM_JAVA_STATIC |
        TM_JAVA_ABSTRACT |
        TM_JAVA_PRIVATE
end
```

Appendix D

ANTLR Grammars

D.1 Grammar file for the first version

```
1   header {
    package translator.syntaticanalyzer;
3   import translator.lib.*;
    import translator.rslast.*;
    import translator.rsllib.*;
6   import java.util.*;
    }
    options {language="Java";}
9
    class RSLParser extends Parser;

12  options {
        k=2;
    }
15
    tokens {
        SCHEME      = "scheme";
18      CLASS       = "class";
        EXTEND      = "extend";
        WITH        = "with";
21      TYPE        = "type";
        VALUE       = "value";
        TEST_CASE   = "test_case";
24      IS          = "is";
        END         = "end";
```

```
        IF          = "if";
27      THEN        = "then";
        ELSIF       = "elsif";
        ELSE        = "else";
30      CASE        = "case";
        OF          = "of";

33      LIST        = "list";
        BOOLEAN     = "Bool";
        INT         = "Int";
36      NAT         = "Nat";
        REAL        = "Real";
        CHAR        = "Char";
39      TEXT        = "Text";
        UNIT        = "Unit";

42
        FALSE       = "false";
        TRUE        = "true";
45
        HD          = "hd";
        TL          = "tl";
48  }

    rslast returns [RSLAst rslast] {rslast = null; LibModule lm = null;}
51      :   lm = lib_module {rslast = new RSLAst(lm);}
        ;

54  lib_module returns [LibModule lm]
    {
        lm = null;
57      SchemeDef sd = null;
        Id identifier = null;
        RSLListDefault<Id> contextList =
60          new RSLListDefault<Id>();
    }
        :   (
63              identifier = id {contextList.getList().add(identifier);}
                (
                    COMMA identifier = id
66                      {contextList.getList().add(identifier);}
```

```
            )
        *)?
69      sd = scheme_def {lm = new LibModule(contextList, sd);}
    ;

72  scheme_def returns [SchemeDef sd]   {
                                        sd = null;
                                        ClassExpr ce = null;
75                                      Id identifier = null;
    }
    :   SCHEME identifier = id EQUAL ce = class_expr
78      {sd = new SchemeDef(identifier, ce);}
    ;

81  class_expr returns [ClassExpr ce] {ce = null;}
    :   ce = basic_class_expr
    |   ce = extending_class_expr
84  |   ce = scheme_instantiation
    ;

87  basic_class_expr returns [BasicClassExpr be]
    {
        be = null;
90      RSLListDefault<Decl> decl_list = new RSLListDefault<Decl>();
        Decl d;
    }
93      :   CLASS (d = decl {decl_list.getList().add(d);})* END
            {be = new BasicClassExpr(decl_list);}
        ;
96
    extending_class_expr returns [ExtendingClassExpr ece]
    {
99      ece = null;
        ClassExpr ce1 = null;
        ClassExpr ce2 = null;
102 }
        :   EXTEND ce1 = class_expr WITH ce2 = class_expr
            {ece = new ExtendingClassExpr(ce1, ce2);}
105     ;

    scheme_instantiation returns [SchemeInstantiation si]
```

```
108 {
        si = null;
        Id identifier = null;
111 }
        :   identifier = id
            {si = new SchemeInstantiation(identifier);}
114     ;

    decl returns [Decl d] {d = null;}
117     :   VALUE d = value_decl
        |   TYPE d = type_decl
        |   TEST_CASE d = test_decl
120     ;

    /*Type Declaration*/
123 type_decl returns [TypeDecl td]
    {
        td = null;
126     RSLListDefault<TypeDef> type_def_list =
          new RSLListDefault<TypeDef>();
        TypeDef def;
129 }
        :   def = type_def
                    {type_def_list.getList().add(def);}
132               (
                    COMMA def = type_def
                    {type_def_list.getList().add(def);}
135               )*
            {td = new TypeDecl(type_def_list);}
        ;
138
    type_def returns [TypeDef td]
    {
141     td = null;
    }
        :   td = sort_def
144     |   td = variant_def
        |   td = short_record_def
        ;
147
    sort_def returns [SortDef sd]
```

```
    {
150     sd = null;
        Id identifier = null;
    }
153     :   identifier = id
            {sd = new SortDef(identifier);}
        ;
156
    variant_def returns [VariantDef vd]
    {   vd = null;
159     Id identifier = null;
        Id identifier2 = null;
        Constructor constructor = null;
162     RSLListDefault<Variant> variant_list = new RSLListDefault<Variant>();
        RSLListDefault<ComponentKind> componentKindList = null;
    }
165     :   identifier = id EQUAL EQUAL identifier2 = id
            {constructor = new Constructor(identifier2);}
            (LPAREN componentKindList = component_kind_list RPAREN)?
168         {
                if(componentKindList == null) {
                    variant_list.getList().add(constructor);
171             } else {
                    variant_list.getList().add(
                        new RecordVariant(constructor, componentKindList));
174                 componentKindList = null;
                }
            }
177         (BAR
                identifier2 = id
                    {constructor = new Constructor(identifier2);}
180             (LPAREN componentKindList = component_kind_list RPAREN)?
                {
                    if(componentKindList == null) {
183                     variant_list.getList().add(constructor);
                    } else {
                        variant_list.getList().add(
186                         new RecordVariant(constructor, componentKindList));
                        componentKindList = null;
                    }
189             }
```

```
            )*
            {vd = new VariantDef(identifier, variant_list);}
192         ;

    short_record_def returns [ShortRecordDef srd]
195 {   srd = null;
        Id identifier = null;
        RSLListDefault<ComponentKind> componentKindString = null;
198 }
        :   identifier = id COLON COLON
                componentKindString = component_kind_string
201         {srd = new ShortRecordDef(identifier, componentKindString);}
        ;

204 component_kind_list returns [RSLListDefault<ComponentKind>
                                componentKindList]
    {
207     componentKindList = new RSLListDefault<ComponentKind>();
        Id identifier1 = null;
        Id identifier2 = null;
210     TypeExpr typeExpr = null;
        TypeExpr typeExpr2 = null;
    }
213     :
            (identifier1 = id COLON)?
                typeExpr = type_expr (GT MINUS LT identifier2 = id)?
216         {
                componentKindList.getList().add(
                new ComponentKind(
219                 new Destructor( identifier1),
                    typeExpr,
                    identifier2));
222             identifier1 = null;
                identifier2 = null;
            }
225         (COMMA (identifier1 = id COLON)?
                typeExpr2 = type_expr (GT MINUS LT identifier2 = id)?
                {
228                 componentKindList.getList().add(
                        new ComponentKind(
                            new Destructor( identifier1),
```

```
231                                             typeExpr2,
                                                identifier2));
                    identifier1 = null;
234                 identifier2 = null;
                }
            )*
237     ;

    component_kind_string returns [RSLListDefault<ComponentKind>
240                                     componentKindString]
    {
        componentKindString = new RSLListDefault<ComponentKind>();
243     Id identifier1 = null;
        Id identifier2 = null;
        TypeExpr typeExpr = null;
246 }
        :
            ((identifier1 = id COLON)? typeExpr = type_expr
249             (GT MINUS LT identifier2 = id)?
                {
                    componentKindString.getList().add(
252                     new ComponentKind(
                            new Destructor(identifier1),
                            typeExpr,
255                         identifier2));
                    identifier1 = null;
                    identifier2 = null;
258             }
            )+
        ;
261

    /*Value Declaration*/
264 value_decl returns [ValueDecl vd]
    {
        vd = null;
267     RSLListDefault<ValueDef> value_def_list =
            new RSLListDefault<ValueDef>();
        ValueDef def;
270 }
        :   def = value_def {value_def_list.getList().add(def);}
```

```
                (COMMA def = value_def
273                 {value_def_list.getList().add(def);}
                )*
            {vd = new ValueDecl(value_def_list);}
276     ;

    value_def returns [ValueDef vd] {vd = null;}
279     :   vd = explicit_function_def
        ;

282 explicit_function_def returns [ExplicitFunctionDef efd]
    {
        efd = null;
285     SingleTyping st = null;
        FormalFunctionApplication ffa = null;
        ValueExpr ve = null;
288     OptionalPrecondition pc = new NoPrecondition();
    }
        :   st = single_typing
291         ffa = formal_function_application "is"
            ve = value_expr
            (pc = precondition)?
294         {efd = new ExplicitFunctionDef(st, ffa, ve, pc);}
        ;

297 formal_function_application returns [FormalFunctionApplication ffa]
    {
        ffa = null;
300 }
        :   ffa = id_application
        ;
303
    id_application returns [IdApplication ia]
    {
306     ia = null;
        Id identifier = null;
        RSLListDefault<FormalFunctionParameter> ffp_list =
309         new RSLListDefault<FormalFunctionParameter>();
        FormalFunctionParameter ffp = null;
    }
312     :   identifier = id
```

```
                (ffp = formal_function_parameter
                {
315                 if(ffp != null)
                        ffp_list.getList().add(ffp);}
                )*
318         {ia =  new IdApplication(identifier, ffp_list);}
        ;

321 formal_function_parameter returns [FormalFunctionParameter ffp]
    {
        ffp = null;
324     RSLListDefault<Binding> binding_list = new RSLListDefault<Binding>();
        Binding b = null;
    }
327     :   LPAREN
            (b = binding
                {binding_list.getList().add(b);}
330             (COMMA b = binding {binding_list.getList().add(b);})*
            )?
            RPAREN
333         {
                if(!binding_list.getList().isEmpty())
                    ffp = new FormalFunctionParameter(binding_list);
336         }
        ;

339 precondition returns [Precondition pc] {pc = null;}
        :   "pre" value_expr
        ;
342
    single_typing returns [SingleTyping st]
    {
345     st = null;
        Binding b = null;
        TypeExpr te = null;
348 }
        :   b = binding COLON te = type_expr
            {st = new SingleTyping(b, te);}
351     ;

    binding returns [Binding b] {b = null;}
```

```
354         :   b = id
            ;

357     /*Test Declarations*/
        test_decl returns [TestDecl td]
        {
360         td = null;
            RSLListDefault<TestDef> test_def_list =
                new RSLListDefault<TestDef>();
363         TestDef def;
        }
            :   def = test_def
366             {test_def_list.getList().add(def);}
                (COMMA def = test_def
                    {test_def_list.getList().add(def);}
369             )*
                {td = new TestDecl(test_def_list);}
            ;
372
        test_def returns [TestDef td]   {   td = null;
                                            Id identifier = null;
375                                         ValueExpr ve = null;
                                        }
            :   LBRACKET identifier = id RBRACKET ve = value_expr
378             {td = new TestDef(identifier, ve);}
            ;

381     /*Type Expression*/
        type_expr returns [TypeExpr te]
        {
384         te = null;
            FunctionArrow fa = null;
            TypeExpr te2 = null;
387         TypeExpr te3 = null;
        }
            :   te2 = type_expr_pr2
390             (
                    (
                        MINUS LT
393                         {fa = FunctionArrow.TOTAL_FUNCTION_ARROW;}
                    |   MINUS DASH MINUS LT
```

```
                        {fa = FunctionArrow.PARTIAL_FUNCTION_ARROW;}
396             )
                te3 = type_expr_pr2
                    {te = new FunctionTypeExpr(
399                         te2,
                            fa,
                            new FunctionResultDescription(
402                             new NoAccessDescription(), te3));
                    }
            )?
405         {if(te == null) te = te2;}
        ;

408 type_expr_pr2 returns [TypeExpr te]
    {
        te = null;
411     TypeExpr te2 = null;
        TypeExpr te3 = null;
        RSLListDefault<TypeExpr> typeExprList =
414         new RSLListDefault<TypeExpr>();}
        : te2 = type_expr_pr1
            {typeExprList.getList().add(te2);}
417     ( LT GT te3 = type_expr_pr1
                {typeExprList.getList().add(te3);}
        )*
420         {
                if(te3 != null) {
                    te = new ProductTypeExpr(typeExprList);
423             }
                else {
                    te = te2;
426             }
            }
        ;
429
    type_expr_pr1 returns [TypeExpr te]
    {
432     te = null;
        TypeExpr te2 = null;
    }
435     : te2 = type_expr_primary
```

```
            (MINUS LIST
                {te = new FiniteListTypeExpr(te2);}
438         )?
          {if(te == null) te = te2;}
        ;
441
    type_expr_primary returns [TypeExpr te]
    {
444     te = null;
        Id identifier = null;
    }
447     :   te = type_literal
        |   LPAREN te = type_expr RPAREN
        |   identifier = id {te = new TypeName(identifier);}
450     ;

    type_literal returns [TypeLiteral tl] {tl = null;}
453     :   INT     {tl = TypeLiteral.RSLINT;}
        |   REAL    {tl = TypeLiteral.RSLREAL;}
        |   NAT     {tl = TypeLiteral.RSLNAT;}
456     |   BOOLEAN {tl = TypeLiteral.RSLBOOL;}
        |   CHAR    {tl = TypeLiteral.RSLCHAR;}
        |   TEXT    {tl = TypeLiteral.RSLTEXT;}
459     |   UNIT    {tl = TypeLiteral.RSLUNIT;}
        ;


462
    /*Value Expressions */
    value_expr returns [ValueExpr ve] {ve = null;}
465     :   ve = infix_expr_pr6
        ;

468 infix_expr_pr6 returns [ValueExpr ve]
    {
        ve = null;
471     ValueExpr ve1 = null;
        RSLInfixOp rio = null;
        ValueExpr ve2 = null;
474     ValueExpr ve3 = null;
    }
        :   ve1 = infix_expr_pr5
```

```
477             (
                    rio = infix_op_pr6
                    ve2 = infix_expr_pr5
480                     {
                            ve3 = ve1;
                            ve1 = new ValueInfixExpr(ve3, rio, ve2);}
483             )*
                {ve = ve1;}
        ;
486
    infix_expr_pr5 returns [ValueExpr ve]
    {
489     ve = null;
        ValueExpr ve1 = null;
        RSLInfixOp rio = null;
492     ValueExpr ve2 = null;
        ValueExpr ve3 = null;
    }
495     :   ve1 = infix_expr_pr4
            (
                rio = infix_op_pr5
498             ve2 = infix_expr_pr4
                    {ve3 = ve1; ve1 = new ValueInfixExpr(ve3, rio, ve2);}
            )*
501         {ve = ve1;}
        ;

504 infix_expr_pr4 returns [ValueExpr ve]
    {
        ve =  null;
507     ValueExpr ve1 = null;
        RSLInfixOp rio = null;
        ValueExpr ve2 = null;
510     ValueExpr ve3 = null;
    }
        :   ve1 = disamb_expr
513         (
                rio = infix_op_pr4
                ve2 = disamb_expr
516                 {ve3 = ve1; ve1 = new ValueInfixExpr(ve3, rio, ve2);}
            )*
```

```
            {ve = ve1;}
519         ;

    disamb_expr returns [ValueExpr ve]
522 {
        ve = null;
        ValueExpr ve1 = null;
525     TypeExpr te = null;
    }
        :   ve1 = prefix_expr
528         (
                COLON te = type_expr
            )?
531         {
                if(te != null) {
                    ve = new DisambiguationExpr(ve1, te);
534             }
                else {
                    ve = ve1;
537             }
            }
        ;
540
    prefix_expr returns [ValueExpr ve]
    {
543     ve = null;
        RSLPrefixOp po = null;
        ValueExpr pe = null;
546 }
        :   po = prefix_op pe = prefix_expr
                {ve = new ValuePrefixExpr(po, pe);}
549     |   ve = primary_value_expr
        ;

552 primary_value_expr returns [ValueExpr ve]
    {
        ve = null;
555     ValueExpr condition = null;
        ValueExpr condition2 = null;
        RSLListDefault<ElsifBranch> elsif_branch_list =
558         new RSLListDefault<ElsifBranch>();
```

```
        RSLListDefault<ValueExpr> optional_value_expr_list =
            new RSLListDefault<ValueExpr>();
561     RSLListDefault<CaseBranch> caseBranchList =
            new RSLListDefault<CaseBranch>();
        ElseBranch eb = null;
564     ValueExpr ve1 = null;
        ValueExpr ve2 = null;
        Pattern p = null;
567 }
        :   LPAREN ve = value_expr RPAREN
        |   IF condition = value_expr THEN
570         ve1 = value_expr
            (
                ELSIF condition2 = value_expr THEN ve2 = value_expr
573                 {elsif_branch_list.getList().add(
                        new ElsifBranch(condition2, ve2));
                    }
576         )*
            (
                ELSE ve2 = value_expr
579                 {eb = new ElseBranch(ve2);}
            )? END
            {ve = new IfExpr(condition, ve1, elsif_branch_list, eb);}
582     |   ve = value_literal
        |   ve1 = name
                LPAREN
585             (
                    ve2 = value_expr
                        {optional_value_expr_list.getList().add(ve2);}
588                 (COMMA ve2 = value_expr
                        {optional_value_expr_list.getList().add(ve2);}
                    )*
591             )?
                RPAREN
            {ve = new ApplicationExpr(ve1, optional_value_expr_list);}
594     |   ve = name
        |   GT DOT
            (
597             ve2 = value_expr
                    {optional_value_expr_list.getList().add(ve2);}
                (COMMA ve2 = value_expr
```

```
600                     {optional_value_expr_list.getList().add(ve2);}
                    )*
                )?
603             DOT LT
                {ve = new EnumeratedListExpr(optional_value_expr_list);}
            |   CASE ve1 = value_expr OF
606             p = pattern MINUS LT ve2 = value_expr
                    {caseBranchList.getList().add(new CaseBranch(p, ve2));}
                (
609                 COMMA p = pattern MINUS LT ve2 = value_expr
                    {caseBranchList.getList().add(new CaseBranch(p, ve2));}
                )*
612             END {ve = new CaseExpr(ve1, caseBranchList);}
            ;

615     value_literal returns [ValueLiteral vl] {vl = null;}
            :   vl = int_lit
            |   vl = bool_lit
618         |   vl = real_lit
            |   vl = text_lit
            |   vl = char_lit
621         ;

        name returns [ValueOrVariableName vn] {vn = null; Id qi = null;}
624         :   qi = qualified_id {vn = new ValueOrVariableName(qi);}
            ;

627     infix_op_pr6 returns [RSLInfixOp rio] {rio = null;}
            :   EQUAL {rio = RSLInfixOp.RSL_INFIX_OP_EQUALS;}
            ;
630
        infix_op_pr5 returns [RSLInfixOp rio] {rio = null;}
            :   PLUS {rio = RSLInfixOp.RSL_INFIX_OP_PLUS;}
633         |   HAT {rio = RSLInfixOp.RSL_INFIX_OP_HAT;}
            ;

636     infix_op_pr4 returns [RSLInfixOp rio] {rio = null;}
            :   STAR {rio = RSLInfixOp.RSL_INFIX_OP_STAR;}
            ;
639
        prefix_op returns [RSLPrefixOp rpo] {rpo = null;}
```

```
        :   HD {rpo = RSLPrefixOp.RSL_PREFIX_OP_HD;}
642     |   TL {rpo = RSLPrefixOp.RSL_PREFIX_OP_TL;}
        ;

645 qualified_id returns [Id qi] {qi = null;}
        :   qi = id
        ;
648
    id returns [Id i] {i = null;}
        : a:IDENT {i = new Id(a.getText());}
651     ;

    pattern returns[Pattern p]
654 {
        p = null;
        ValueOrVariableName vovn = null;
657     ValueOrVariableName vovn2 = null;
        ValueOrVariableName vovn3 = null;
        ValueLiteral vl = null;
660     RSLListDefault<Pattern> innerPatternList =
            new RSLListDefault<Pattern>();
    }
663     :   vl = value_literal
            {p = new ValueLiteralPattern(vl);}
        |   vovn = name
666         {p = new NamePattern(vovn.getId());}
        |   vovn = name
            LPAREN
669             vovn2 = name
                    {innerPatternList.getList().add(
                        new NamePattern(vovn2.getId()));}
672         (
                COMMA vovn3 = name
                    {innerPatternList.getList().add(
675                     new NamePattern(vovn3.getId()));}
            )*
            RPAREN
678         {p = new RecordPattern(vovn, innerPatternList );}
        |   UNDERSCORE
            {p = new WildcardPattern();}
681     ;
```

```
    int_lit returns [ValueLiteralInteger i] {i = null;}
684     :   a:INTEGER_LITERAL {i = new ValueLiteralInteger(a.getText());}
        ;

687 real_lit returns [ValueLiteralReal r] {r = null;}
        :   a:REAL_LITERAL {r = new ValueLiteralReal(a.getText());}
        ;
690
    bool_lit returns [ValueLiteralBool b] {b = null;}
        :   FALSE {b = new ValueLiteralBool("false");}
693     |   TRUE  {b = new ValueLiteralBool("true");}
        ;

696 text_lit returns [ValueLiteralText t] {t = null;}
        :   a:TEXT_LITERAL {t = new ValueLiteralText(a.getText());}
        ;
699
    char_lit returns [ValueLiteralChar c] {c = null;}
        :   a:CHAR_LITERAL {c = new ValueLiteralChar(a.getText());}
702     ;

    class RSLLexer extends Lexer;
705 options {
        charVocabulary='\u0000'..'\u007F'; //Allow only ascii
        k=2;
708     testLiterals=false;
    }

711 COMMA           : ",";
    COLON           : ":";

714 LPAREN          : "(";
    RPAREN          : ")";
    LBRACKET        : "[";
717 RBRACKET        : "]";
    PLUS            : "+";
    MINUS           : "-";
720 STAR            : "*";
    HAT             : "^";
    EQUAL           : "=";
```

```
723   NOT_EQUAL       : "~=";
      LT              : ">";
      GT              : "<";
726   CURLYDASH       : "~";
      BAR             : "|";
      UNDERSCORE      : "_";
729   //DOT             : ".";


732   CHAR_LITERAL    : '\''! (ESC|~('\''|'\\')) '\''!;

      TEXT_LITERAL    :       '"'! (ESC|~('"'|'\\'))* '"'!;
735

      protected
738   ESC
              :       '\\'
                      (       'n'
741                   |       'r'
                      |       't'
                      |       'b'
744                   |       'f'
                      |       '"'
                      |       '\''
747                   |       '\\'
                      |       '0'..'3'
                              (
750                                   options {
                                              warnWhenFollowAmbig = false;
                                      }
753                           :       '0'..'7'
                                      (
                                              options {
756                                                   warnWhenFollowAmbig = false;
                                              }
                                      :       '0'..'7'
759                                   )?
                              )?
                      |       '4'..'7'
762                           (
                                      options {
```

```
                                                warnWhenFollowAmbig = false;
765                                     }
                                :       '0'..'7'
                                )?
768                 )
            ;

771 // a numeric literal
    INTEGER_LITERAL
        :   '.' {_ttype = DOT;}
774         |
            (
                '0'
777             // special case for just '0'
                    |
                ('1'..'9') ('0'..'9')*
780             // non-zero decimal
                    )
                    (
783             ('.' ('0'..'9')+) => ('.' ('0'..'9')+)
                {_ttype = REAL_LITERAL;}
            )?
786         ;


789 IDENT
            options {testLiterals=true;}
            :       ('a'..'z'|'A'..'Z')
792         ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
            ;

795

    WS
798     :   (   ' '
            |   '\r' '\n' {newline();}
            |   '\n' {newline();}
801         |   '\t'
            )
            {$setType(Token.SKIP);}
804     ;
```

```
    // multiple-line comments
807 ML_COMMENT
            :       "/*"
                    (
810 options {
            generateAmbigWarnings=false;
    }
813             :
                            { LA(2)!='/' }? '*'
                    |       '\r' '\n'                {newline();}
816                 |       '\r'                     {newline();}
                    |       '\n'                     {newline();}
                    |       ~('*'|'\n'|'\r')
819                 )*
                    "*/"
                    {$setType(Token.SKIP);}
822             ;
```

D.2 Grammar file for the second version

```
 1  header {
    package translator.syntacticanalyzer;
 3  import translator.lib.*;
    import translator.rslast2.*;
    import translator.rsllib.*;
 6  import java.util.*;
    }
    options {language="Java";}
 9
    class RSLParser extends Parser;

12  options {
        k=2;
    }
15
    tokens {
        SCHEME      = "scheme";
18      CLASS       = "class";
        EXTEND      = "extend";
```

```
        WITH        = "with";
21      TYPE        = "type";
        VALUE       = "value";
        TEST_CASE   = "test_case";
24      IS          = "is";
        END         = "end";
        IF          = "if";
27      THEN        = "then";
        ELSIF       = "elsif";
        ELSE        = "else";
30      CASE        = "case";
        OF          = "of";

33      LIST        = "list";
        BOOLEAN     = "Bool";
        INT         = "Int";
36      NAT         = "Nat";
        REAL        = "Real";
        CHAR        = "Char";
39      TEXT        = "Text";
        UNIT        = "Unit";

42
        FALSE       = "false";
        TRUE        = "true";
45
        HD          = "hd";
        TL          = "tl";
48  }

    rslast returns [TM_RSLAst rslast]
51  {
        rslast = null;
        TM_LibModule lm = null;
54  }
        :   lm = lib_module {rslast = new TM_RSLAst(lm);}
        ;
57
    lib_module returns [TM_LibModule lm]
    {
60      lm = null;
```

```
        TM_SchemeDef sd = null;
        TM_Id identifier = null;
63      RSLListDefault<TM_Id> contextList =
            new RSLListDefault<TM_Id>();
    }
66      :   (
                identifier = id {contextList.getList().add(identifier);}
                (
69                  COMMA identifier = id
                        {contextList.getList().add(identifier);}
                )*
72          )?
            sd = scheme_def
                {lm = new TM_LibModule(contextList, sd);}
75
        ;

78  scheme_def returns [TM_SchemeDef sd]   {
                                            sd = null;
                                            TM_ClassExpr ce = null;
81                                          TM_Id identifier = null;
                                        }
        :   SCHEME identifier = id EQUAL ce = class_expr
84          {sd = new TM_SchemeDef(identifier, ce);}
        ;

87  class_expr returns [TM_ClassExpr ce] {ce = null;}
        :   ce = basic_class_expr
        |   ce = extending_class_expr
90      |   ce = scheme_instantiation
        ;

93  basic_class_expr returns [TM_BasicClassExpr be]
    {
        be = null;
96      RSLListDefault<TM_Decl> decl_list =
            new RSLListDefault<TM_Decl>(); TM_Decl d;}
        :   CLASS
99          (
                d = decl
                    {decl_list.getList().add(d);}
```

```
            )*
            END
            {be = new TM_BasicClassExpr(decl_list);}
        ;

    extending_class_expr returns [TM_ExtendingClassExpr ece]
    {
        ece = null;
        TM_ClassExpr ce1 = null;
        TM_ClassExpr ce2 = null;
    }
        :   EXTEND ce1 = class_expr WITH ce2 = class_expr
            {ece = new TM_ExtendingClassExpr(ce1, ce2);}
        ;

    scheme_instantiation returns [TM_SchemeInstantiation si]
    {
        si = null;
        TM_Id identifier = null;
    }
        :   identifier = id
            {si = new TM_SchemeInstantiation(identifier);}
        ;


    decl returns [TM_Decl d] {d = null;}
        :   VALUE d = value_decl
        |   TYPE d = type_decl
        |   TEST_CASE d = test_decl
        ;


    //Type Declaration
    type_decl returns [TM_TypeDecl td]
    {
        td = null;
        RSLListDefault<TM_TypeDef> type_def_list =
            new RSLListDefault<TM_TypeDef>();
        TM_TypeDef def;
    }
        :   def = type_def
```

```
                {type_def_list.getList().add(def);}
144             (COMMA def = type_def
                {type_def_list.getList().add(def);}
                )*
147             {td = new TM_TypeDecl(type_def_list);}
            ;

150     type_def returns [TM_TypeDef td] {td = null;}
            :   td = sort_def
            |   td = variant_def
153         |   td = short_record_def
            ;

156     sort_def returns [TM_SortDef sd] {sd = null; TM_Id identifier = null;}
            :   identifier = id
                {sd = new TM_SortDef(identifier);}
159         ;

        variant_def returns [TM_VariantDef vd]
162     {
            vd = null;
            TM_Id identifier = null;
165         TM_Id identifier2 = null;
            TM_Constructor constructor = null;
            RSLListDefault<TM_Variant> variant_list =
168             new RSLListDefault<TM_Variant>();
            RSLListDefault<TM_ComponentKind>
                componentKindList = null;
171     }
            :   identifier = id EQUAL EQUAL
                identifier2 = id
174                 {constructor = new TM_Constructor(identifier2);}
                (LPAREN componentKindList = component_kind_list RPAREN)?
                {
177                 if(componentKindList == null) {
                        variant_list.getList().add(
                            new Make_TM_Constructor(constructor));
180                 } else {
                        variant_list.getList().add(
                            new TM_RecordVariant(
183                             constructor,
```

```
                            componentKindList
                        ));
186                 componentKindList = null;
                }
            }
189         (BAR
                identifier2 = id
                    {constructor = new TM_Constructor(identifier2);}
192             (
                    LPAREN
                        componentKindList = component_kind_list
195                 RPAREN
                )?
                {
198                 if(componentKindList == null) {
                        variant_list.getList().add(
                            new Make_TM_Constructor(constructor));
201                 } else {
                        variant_list.getList().add(
                            new TM_RecordVariant(
204                             constructor,
                                componentKindList
                            ));
207                 componentKindList = null;
                }
            }
210         )*
            {vd = new TM_VariantDef(identifier, variant_list);}
        ;
213
    short_record_def returns [TM_ShortRecordDef srd]
    {
216     srd = null;
        TM_Id identifier = null;
        RSLListDefault<TM_ComponentKind> componentKindString = null;
219 }
        :   identifier = id COLON COLON
            componentKindString = component_kind_string
222             {srd = new TM_ShortRecordDef(identifier, componentKindString);}
        ;
```

```
225  component_kind_list returns
     [RSLListDefault<TM_ComponentKind> componentKindList]
     {
228      componentKindList = new RSLListDefault<TM_ComponentKind>();
         TM_Id identifier1 = null;
         TM_Id identifier2 = null;
231      TM_TypeExpr typeExpr = null;
         TM_TypeExpr typeExpr2 = null;
     }
234      :
             (identifier1 = id COLON)?
             typeExpr = type_expr
237          (GT MINUS LT identifier2 = id)?
             {
                 componentKindList.getList().add(
240                  new TM_ComponentKind(
                         new TM_Destructor(identifier1),
                         typeExpr,
243                      (identifier2 != null ?
                             new TM_Reconstructor(identifier2) :
                             new TM_NoReconstructor())));
246              identifier1 = null;
                 identifier2 = null;
             }
249          (
                 COMMA
                 (identifier1 = id COLON)?
252              typeExpr2 = type_expr
                 (GT MINUS LT identifier2 = id)?
                 {
255                  componentKindList.getList().add(
                         new TM_ComponentKind(
                             new TM_Destructor(identifier1),
258                          typeExpr2,
                             (identifier2 != null ?
                                 new TM_Reconstructor(identifier2) :
261                              new TM_NoReconstructor()))
                             );
                     identifier1 = null;
264                  identifier2 = null;
                 }
```

```
            )*
267         ;

    component_kind_string returns
270 [RSLListDefault<TM_ComponentKind> componentKindString]
    {
        componentKindString = new RSLListDefault<TM_ComponentKind>();
273     TM_Id identifier1 = null;
        TM_Id identifier2 = null;
        TM_TypeExpr typeExpr = null;
276 }
        :
            (
279             (identifier1 = id COLON)?
                typeExpr = type_expr
                (GT MINUS LT identifier2 = id)?
282             {
                    componentKindString.getList().add(
                        new TM_ComponentKind(
285                         new TM_Destructor(identifier1),
                            typeExpr,
                            identifier2 != null ?
288                             new TM_Reconstructor(identifier2) :
                                new TM_NoReconstructor())
                    );
291                 identifier1 = null;
                    identifier2 = null;
                }
294         )+
        ;

297 //Value Declaration
    value_decl returns [TM_ValueDecl vd]
    {
300     vd = null;
        RSLListDefault<TM_ValueDef> value_def_list =
            new RSLListDefault<TM_ValueDef>();
303     TM_ValueDef def;
    }
        :   def = value_def
306             {value_def_list.getList().add(def);}
```

```
            (COMMA def = value_def
                {value_def_list.getList().add(def);})*
309         {vd = new TM_ValueDecl(value_def_list);}
        ;

312 value_def returns [TM_ValueDef vd] {vd = null;}
        :   vd = explicit_function_def
        ;
315
    explicit_function_def returns [TM_ExplicitFunctionDef efd]
    {
318     efd = null;
        TM_SingleTyping st = null;
        TM_IdApplication ia = null;
321     TM_ValueExpr ve = null;
    }
        :   st = single_typing ia = id_application IS ve = value_expr
324         {efd = new TM_ExplicitFunctionDef(st, ia, ve);}
        ;

327 id_application returns [TM_IdApplication ia]
    {
        ia = null;
330     TM_Id identifier = null;
        RSLListDefault<TM_FormalFunctionParameter> ffp_list =
            new RSLListDefault<TM_FormalFunctionParameter>();
333     TM_FormalFunctionParameter ffp = null;
    }
        :   identifier = id
336         (ffp = formal_function_parameter
                {if(ffp != null) ffp_list.getList().add(ffp);}
            )*
339         {ia =  new TM_IdApplication(identifier, ffp_list);}
        ;

342 formal_function_parameter returns [TM_FormalFunctionParameter ffp]
    {
        ffp = null;
345     RSLListDefault<TM_Binding> binding_list =
            new RSLListDefault<TM_Binding>();
        TM_Binding b = null;
```

```
348 }
        :   LPAREN
            (b = binding
351             {binding_list.getList().add(b);}
            (COMMA b = binding
                {binding_list.getList().add(b);}
354         )*
            )?
            RPAREN
357         {
                if(!binding_list.getList().isEmpty())
                    ffp = new TM_FormalFunctionParameter(binding_list);
360         }
        ;

363
    single_typing returns [TM_SingleTyping st]
    {
366     st = null;
        TM_Binding b = null;
        TM_TypeExpr te = null;}
369     :   b = binding COLON te = type_expr
            {st = new TM_SingleTyping(b, te);}
        ;
372

    binding returns [TM_Binding b] {b = null; TM_Id identifier = null;}
375     :   identifier = id {b = new TM_Binding(identifier);}
        ;

378
    //Test Declarations
    test_decl returns [TM_TestDecl td]
381 {
        td = null;
        RSLListDefault<TM_TestDef> test_def_list =
384         new RSLListDefault<TM_TestDef>();
        TM_TestDef def;
    }
387     :   def = test_def {test_def_list.getList().add(def);}
            (COMMA def = test_def
```

```
                {test_def_list.getList().add(def);}
390         )*
            {td = new TM_TestDecl(test_def_list);}
        ;
393
    test_def returns [TM_TestCase td]
    {
396     td = null;
        TM_Id identifier = null;
        TM_ValueExpr ve = null;
399 }
        :   LBRACKET identifier = id RBRACKET ve = value_expr
            {td = new TM_TestCase(identifier, ve);}
402     ;

    //Type Expression
405 type_expr returns [TM_TypeExpr te]
    {
        te = null;
408     TM_FunctionArrow fa = null;
        TM_TypeExpr te2 = null;
        TM_TypeExpr te3 = null;}
411     :   te2 = type_expr_pr2
            (
                (MINUS LT {fa = new TM_TOTAL_FUNCTION_ARROW();})
414             te3 = type_expr_pr2
                    {te = new TM_FunctionTypeExpr(te2, fa, te3);}
            )?
417         {if(te == null) te = te2;}
        ;

420 type_expr_pr2 returns [TM_TypeExpr te]
    {
        te = null;
423     TM_TypeExpr te2 = null;
        TM_TypeExpr te3 = null;
        RSLListDefault<TM_TypeExpr> typeExprList =
426         new RSLListDefault<TM_TypeExpr>();
    }
        : te2 = type_expr_pr1
429         {typeExprList.getList().add(te2);}
```

```
        ( LT GT te3 = type_expr_pr1
              {typeExprList.getList().add(te3);}
        )*
          {
              if(te3 != null) {
                  te = new TM_TypeExprProduct(typeExprList);
              }
              else {
                  te = te2;
              }
          }
    ;

type_expr_pr1 returns [TM_TypeExpr te]
{
    te = null;
    TM_TypeExpr te2 = null;
}
    : te2 = type_expr_primary
        (MINUS LIST
            {te = new TM_TypeExprList(
                    new TM_FiniteListTypeExpr(te2));}
        )?
        {if(te == null) te = te2;}
    ;

type_expr_primary returns [TM_TypeExpr te]
{
    te = null;
    TM_Id identifier = null;
}
    :   te = type_literal
    |   LPAREN te = type_expr RPAREN
    |   identifier = id {te = new TM_TypeName(identifier);}
    ;

type_literal returns [TM_TypeLiteral tl] {tl = null;}
    :   INT     {tl = new TM_TypeLiteral(new TM_RSL_INT());}
    |   REAL    {tl = new TM_TypeLiteral(new TM_RSL_REAL());}
    |   NAT     {tl = new TM_TypeLiteral(new TM_RSL_NAT());}
    |   BOOLEAN {tl = new TM_TypeLiteral(new TM_RSL_BOOL());}
```

```
471       |   CHAR    {tl = new TM_TypeLiteral(new TM_RSL_CHAR());}
          |   TEXT    {tl = new TM_TypeLiteral(new TM_RSL_TEXT());}
          |   UNIT    {tl = new TM_TypeLiteral(new TM_RSL_UNIT());}
474       ;


477   //Value Expressions
      value_expr returns [TM_ValueExpr ve] {ve = null;}
          :   ve = infix_expr_pr6
480       ;

      infix_expr_pr6 returns [TM_ValueExpr ve]
483   {
          ve = null;
          TM_ValueExpr ve1 = null;
486       TM_InfixOperator rio = null;
          TM_ValueExpr ve2 = null;
          TM_ValueExpr ve3 = null;}
489       :   ve1 = infix_expr_pr5
              (rio = infix_op_pr6 ve2 = infix_expr_pr5
                  {ve3 = ve1; ve1 = new TM_ValueInfixExpr(ve3, rio, ve2);}
492           )*
              {ve = ve1;}
          ;
495
      infix_expr_pr5 returns [TM_ValueExpr ve]
      {
498       ve = null;
          TM_ValueExpr ve1 = null;
          TM_InfixOperator rio = null;
501       TM_ValueExpr ve2 = null;
          TM_ValueExpr ve3 = null;
      }
504       :   ve1 = infix_expr_pr4
              (rio = infix_op_pr5 ve2 = infix_expr_pr4
                  {ve3 = ve1; ve1 = new TM_ValueInfixExpr(ve3, rio, ve2);}
507           )*
              {ve = ve1;}
          ;
510
      infix_expr_pr4 returns [TM_ValueExpr ve]
```

```
    {
513     ve =  null;
        TM_ValueExpr ve1 = null;
        TM_InfixOperator rio = null;
516     TM_ValueExpr ve2 = null;
        TM_ValueExpr ve3 = null;}
        :   ve1 = disamb_expr
519         (rio = infix_op_pr4 ve2 = disamb_expr
                {ve3 = ve1; ve1 = new TM_ValueInfixExpr(ve3, rio, ve2);}
            )*
522         {ve = ve1;}
        ;

525 disamb_expr returns [TM_ValueExpr ve]
    {
        ve = null;
528     TM_ValueExpr ve1 = null;
        TM_TypeExpr te = null;
    }
531     :   ve1 = prefix_expr (COLON te = type_expr)?
            {if(te != null) {ve = ve1;} else {ve = ve1;}}
        ;
534
    prefix_expr returns [TM_ValueExpr ve]
    {
537     ve = null;
        TM_PrefixOperator po = null;
        TM_ValueExpr pe = null;}
540     :   po = prefix_op pe = prefix_expr
                {ve = new TM_ValuePrefixExpr(po, pe);}
        |   ve = primary_value_expr
543     ;

    primary_value_expr returns [TM_ValueExpr ve]
546 {
        ve = null;
        TM_ValueExpr condition = null;
549     TM_ValueExpr condition2 = null;
        RSLListDefault<TM_Elsif> elsif_branch_list =
            new RSLListDefault<TM_Elsif>();
552     RSLListDefault<TM_ValueExpr> optional_value_expr_list =
```

```
            new RSLListDefault<TM_ValueExpr>();
        RSLListDefault<TM_CaseBranch> caseBranchList =
555         new RSLListDefault<TM_CaseBranch>();
        TM_ValueExpr ve1 = null;
        TM_ValueExpr ve2 = null;
558     TM_Pattern p = null;
        TM_ValueLiteral vl = null;
    }
561     :   LPAREN ve1 = value_expr RPAREN
                {ve = new TM_ParenthesizedExpr(ve1);}
        |   IF condition = value_expr THEN ve1 = value_expr
564         (ELSIF condition2 = value_expr THEN ve2 = value_expr
                {elsif_branch_list.getList().add(new TM_Elsif(
                            condition2,
567                         ve2));
                }
            )*
570         (ELSE ve2 = value_expr)? END
            {ve = new Make_TM_IfExpr(
                    new TM_IfExpr(
573                     condition,
                        ve1,
                        elsif_branch_list,
576                     ve2));
            }
        |   vl = value_literal {ve = new Make_TM_ValueLiteral(vl);}
579     |   ve1 = name
            LPAREN
            (
582             ve2 = value_expr
                    {optional_value_expr_list.getList().add(ve2);}
                (
585                 COMMA
                    ve2 = value_expr
                        {optional_value_expr_list.getList().add(ve2);}
588             )*
            )?
            RPAREN
591             {ve = new TM_ApplicationExpr(
                        ve1,
                        optional_value_expr_list);
```

```
594             }
        |   ve = name
        |   GT DOT
597         (ve2 = value_expr
                {optional_value_expr_list.getList().add(ve2);}
            (COMMA ve2 = value_expr
600                 {optional_value_expr_list.getList().add(ve2);}
            )*
            )?
603         DOT LT
                {ve = new Make_TM_ListExpr(
                    new TM_EnumeratedListExpr(optional_value_expr_list));}
606     |   CASE ve1 = value_expr OF
            p = pattern MINUS LT ve2 = value_expr
                {caseBranchList.getList().add(new TM_CaseBranch(p, ve2));}
609         (
                COMMA p = pattern MINUS LT ve2 = value_expr
                {caseBranchList.getList().add(new TM_CaseBranch(p, ve2));}
612         )*
            END {ve = new TM_CaseExpr(ve1, caseBranchList);}
        ;
615
    value_literal returns [TM_ValueLiteral vl] {vl = null;}
        :   vl = int_lit
618     |   vl = bool_lit
        |   vl = real_lit
        |   vl = text_lit
621     |   vl = char_lit
        ;

624 name returns [Make_TM_ValueOrVariableName vn]
    {vn = null; TM_Id qi = null;}
        :   qi = qualified_id
627             {vn = new Make_TM_ValueOrVariableName(
                            new TM_ValueOrVariableName(qi));
                }
630     ;

    infix_op_pr6 returns [TM_InfixOperator rio] {rio = null;}
633     :   EQUAL {rio = new TM_RSL_EQUAL();}
        ;
```

```
636   infix_op_pr5 returns [TM_InfixOperator rio] {rio = null;}
          :   PLUS {rio = new TM_RSL_PLUS();}
          |   HAT {rio = new TM_RSL_HAT();}
639       ;

      infix_op_pr4 returns [TM_InfixOperator rio] {rio = null;}
642       :   STAR {rio = new TM_RSL_STAR();}
          |   SLASH {rio = new TM_RSL_SLASH();}
          ;
645
      prefix_op returns [TM_PrefixOperator rpo] {rpo = null;}
          :   HD {rpo = new TM_RSL_HD();}
648       |   TL {rpo = new TM_RSL_TL();}
          ;

651   qualified_id returns [TM_Id qi] {qi = null;}
          :   qi = id
          ;
654
      id returns [TM_Id i] {i = null;}
          : a:IDENT {i = new TM_Id(a.getText());}
657       ;

      pattern returns[TM_Pattern p]
660   {
          p = null;
          Make_TM_ValueOrVariableName vovn = null;
663       Make_TM_ValueOrVariableName vovn2 = null;
          Make_TM_ValueOrVariableName vovn3 = null;
          TM_ValueLiteral vl = null;
666       RSLListDefault<TM_Pattern> innerPatternList =
              new RSLListDefault<TM_Pattern>();
      }
669       :   vl = value_literal
                  {p = new TM_ValueLiteralPattern(vl);}
          |   vovn = name
672               {p = new TM_NamePattern(
                          vovn.value_or_variable_name().id(),
                          new TM_NoOptionalId());
675               }
```

```
        |   vovn = name
            LPAREN
678         vovn2 = name
                {innerPatternList.getList().add(
                    new TM_NamePattern(
681                     vovn2.value_or_variable_name().id(),
                        new TM_NoOptionalId()));
                }
684         (COMMA vovn3 = name
                {innerPatternList.getList().add(
                        new TM_NamePattern(
687                         vovn3.value_or_variable_name().id(),
                            new TM_NoOptionalId()));}
            )*
690         RPAREN
                {p = new TM_RecordPattern(
                    vovn.value_or_variable_name(),
693                 innerPatternList );
                }
        |   UNDERSCORE {p = new TM_WildcardPattern();}
696     ;

    int_lit returns [TM_ValueLiteralInteger i] {i = null;}
699     :   a:INTEGER_LITERAL
                {i = new TM_ValueLiteralInteger(a.getText());}
        ;
702
    real_lit returns [TM_ValueLiteralReal r] {r = null;}
        :   a:REAL_LITERAL
705             {r = new TM_ValueLiteralReal(a.getText());}
        ;

708 bool_lit returns [TM_ValueLiteralBool b] {b = null;}
        :   FALSE {b = new TM_ValueLiteralBool("false");}
        |   TRUE  {b = new TM_ValueLiteralBool("true");}
711     ;

    text_lit returns [TM_ValueLiteralText t] {t = null;}
714     :   a:TEXT_LITERAL
                {t = new TM_ValueLiteralText(a.getText());}
        ;
```

```
717
    char_lit returns [TM_ValueLiteralChar c] {c = null;}
        :   a:CHAR_LITERAL
720             {c = new TM_ValueLiteralChar(a.getText());}
        ;

723
    class RSLLexer extends Lexer;
    options {
726     charVocabulary='\u0000'..'\u007F'; //Allow only ascii
        k=2;
        testLiterals=false;
729 }

    COMMA           : ",";
732 COLON           : ":";

    LPAREN          : "(";
735 RPAREN          : ")";
    LBRACKET        : "[";
    RBRACKET        : "]";
738 PLUS            : "+";
    MINUS           : "-";
    STAR            : "*";
741 SLASH           : "/";
    HAT             : "^";
    EQUAL           : "=";
744 NOT_EQUAL       : "~=";
    LT              : ">";
    GT              : "<";
747 CURLYDASH       : "~";
    BAR             : "|";
    UNDERSCORE      : "_";
750
        /*DOT is matched by rule for integers and reals.*/
    //DOT             : ".";
753

    CHAR_LITERAL    : '\''! (ESC|~('\''|'\\')) '\''!;
756
```

```
    TEXT_LITERAL
759         :       '"'! (ESC|~('"'|'\\'))* '"'!;

    protected
762 ESC
            :       '\\'
                    (       'r'
765                 |       'n'
                    |       't'
                    |       'a'
768                 |       'b'
            |   'f'
            |   'v'
771         |   '?'
                    |       '\\'
                    |       '\''
774                 |       '"'
                    |       OCT_DIGIT
                            (
777                                 options
                        {warnWhenFollowAmbig = false;}
                            :       OCT_DIGIT
780                             (
                                        options
                                {warnWhenFollowAmbig = false;}
783                             :       OCT_DIGIT
                                )?
                            )?
786         |   'x'
                (
                    options {greedy=true;}
789                 :
                        '0'..'9'|
                        'a'..'f'|
792                     'A'..'F'
                )+
                    )
795         ;

    protected
798 OCT_DIGIT
```

```
        :   '0'..'7'
        ;
801
    // a numeric literal
    INTEGER_LITERAL
804         {boolean isDecimal=false; Token t= null;}
        :   '.' {_ttype = DOT;}
            |
807         (
                '0'
                // special case for just '0'
810                 |
                ('1'..'9') ('0'..'9')*
                // non-zero decimal
813                 )
                    (   ('.' ('0'..'9')+) =>
                ('.' ('0'..'9')+)
816                 {_ttype = REAL_LITERAL;}
            )?
            ;
819

    IDENT
822         options {testLiterals=true;}
            :       ('a'..'z'|'A'..'Z')
            ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
825         ;


828
    WS
        :   (   ' '
831         |   '\r' '\n' {newline();}
            |   '\n' {newline();}
            |   '\t'
834         )
            {$setType(Token.SKIP);}
        ;
837
    // multiple-line comments
    ML_COMMENT
```

```
840         :       "/*"
                    (
                            options {
843                                     generateAmbigWarnings=false;
                            }
                    :
846                         { LA(2)!='/' }? '*'
                    |       '\r' '\n'               {newline();}
                    |       '\r'                    {newline();}
849                 |       '\n'                    {newline();}
                    |       ~('*'|'\n'|'\r')
                    )*
852                 "*/"
                    {$setType(Token.SKIP);}
            ;
```

Appendix E

Source Code, First Version

E.1 translator

E.1.1 Translator.java

```
package translator;

import java.io.*;
import java.util.*;

import translator.lib.*;
import translator.rslast.*;
import translator.rsllib.*;
import translator.javaast.*;
import translator.syntaticanalyzer.*;

public class Translator {
  private static Properties properties = null;
  private static boolean createVisitorMethods = false;
  private static Translator translator = null;
  private static HashMap<Id, HashMap<Id, TypeEvaluator>>
      map =
       new HashMap<Id, HashMap<Id, TypeEvaluator>>();
  private static HashMap<Id, ArrayList<TypeEvaluator>>
      recordMapType =
       new HashMap<Id, ArrayList<TypeEvaluator>>();
  private static HashMap<Id, ArrayList<Id>> recordMapId =
       new HashMap<Id, ArrayList<Id>>();
  private static HashMap<Id, ArrayList<TypeEvaluator>>
      functionMapType =
       new HashMap<Id, ArrayList<TypeEvaluator>>();
  private static HashMap<Id, Id> parentMap = new HashMap<Id
      , Id>();
```

```
25    private static HashMap < Id , TypeEvaluator >
          functionResultType =
          new HashMap < Id , TypeEvaluator > ();
      private static ArrayList < String > getMethodList = new
          ArrayList < String > ();

      private Translator() {
30      properties = new Properties();
        try {
          properties.load(new FileInputStream("default.properties"))
              ;
        }
        catch (FileNotFoundException fnfe) {
35        System.out.println("default.properties file not found");
        }
        catch (IOException ioe) {
          System.out.println("IOException: " + ioe);
        }
40      init();
      }

      private Translator(String propertiesFileName) {
        properties = new Properties();
45      try {
          properties.load(new FileInputStream(propertiesFileName));
        }
        catch (FileNotFoundException fnfe) {
          System.out.println(propertiesFileName + " not found");
50      }
        catch (IOException ioe) {
          System.out.println("IOException: " + ioe);
        }
        init();
55    }

      private void init() {
        createVisitorMethods =
            (new Boolean(getProperty("createVisitorMethods"))).
60          booleanValue();
      }

      public static RSLAst rsl2rslast(Reader reader) {
        RSLAst rslast = null;
65
        try {
          RSLLexer lexer = new RSLLexer(reader);
          RSLParser parser = new RSLParser(lexer);
          rslast = parser.rslast();
70      }
```

```
    catch (Exception e) {
      e.printStackTrace();
    }
    return rslast;
  }

  public static JavaAst rslast2javaast(RSLAst rslast) {
    JavaAst _v0 = null;

    RSLMap<String, CompilationUnit> compilationUnitMap =
        rslast2javaast(rslast.getLibModule().getSchemeDef().
            getClassExpr(),
                       rslast);

    _v0 = new JavaAst(compilationUnitMap);

    return _v0;
  }

  public static RSLMap<String, CompilationUnit>
     rslast2javaast(ClassExpr
      classExpr, RSLAst rslast) {
    RSLMap<String, CompilationUnit> _v0 = null;

    if (classExpr instanceof ExtendingClassExpr) {
      _v0 =
          rslast2javaast( ((ExtendingClassExpr) classExpr).
              getBaseClass(),
                         rslast);
      if (_v0 == null) {
        _v0 =
            rslast2javaast( ((ExtendingClassExpr) classExpr).
                           getExtensionClass(), rslast);
      }
      else {
        _v0 = _v0.union(
            rslast2javaast( ((ExtendingClassExpr) classExpr).
                           getExtensionClass(), rslast)
            );
      }
    }
    else if (classExpr instanceof SchemeInstantiation) {
      if (true) {
        System.out.println("Translating: " +
                           translator.getProperty("extensiondir"
                               ) + "/" +
                           rslId2JavaName( ((
                               SchemeInstantiation)
                                           classExpr).
```

```
115                                             getId()).getText());
          translator.translate(translator.getProperty("
              extensiondir") + "/" +
                               rslId2JavaName( ( (
                                   SchemeInstantiation)
                                                 classExpr).
                                                getId()).getText(),
120                            (new Boolean(translator
                                                .getProperty(
              "writeExtensionFiles")).booleanValue()),
                               (new Boolean(translator.
                                                getProperty(
125           "printJava")).booleanValue()));
        }
      }
      else if (classExpr instanceof BasicClassExpr) {
        if (translator.getProperty("doNotTranslate") == null ||
130         ! (new ArrayList < String > (Arrays.asList(
            translator.getProperty("doNotTranslate").split("$"))).
              contains(rslId2JavaName(rslast.getLibModule().
                                      getSchemeDef().getId()).
                       getText()))) {
135       RSLList < ClassDeclaration > cdList =
            makeClassDeclarationList1( ( (BasicClassExpr)
              classExpr).
                                       getDeclList());

          if (translator.getProperty("getMethods") != null) {
140         getMethodList =
              new ArrayList < String > (Arrays.asList(
              translator.getProperty("getMethods").split("$")
              ));
          }
145       else {
            getMethodList = new ArrayList < String > ();
          }
          TypeDecorateVisitor tdVisitor =
            new TypeDecorateVisitor(map, recordMapType,
              recordMapId,
150                                   functionMapType,
                                        functionResultType,
                                      parentMap, getMethodList);
          rslast.accept(tdVisitor);
          TypeDecorateVisitor tdVisitor2 =
            new TypeDecorateVisitor(map, recordMapType,
155                                   recordMapId, functionMapType
                                        ,
                                      functionResultType,
                                        parentMap,
```

```
                                        getMethodList);
        rslast.accept(tdVisitor2);

        String[] names = new String[cdList.len()];
        CompilationUnit[] files = new CompilationUnit[cdList.len
            ()];
        for (int i = 0; i < cdList.len(); i++) {
          names[i] = cdList.get(i + 1).getSimpleName().getText()
              ;
          files[i] = new CompilationUnit(
              makePackageDeclaration(),
              makeImportDeclarationList(rslast),
              new RSLListDefault < TypeDeclaration > (
              cdList.get(i + 1))
              );
          //System.out.println("Writing: " + names[i]);
        }

        _v0 = new RSLMapDefault < String, CompilationUnit > (
            names, files);
        _v0.union(new RSLMapDefault < String, CompilationUnit
            > (
            rslId2JavaName(rslast.getLibModule().
                           getSchemeDef().getId()).getText(),
            new CompilationUnit(
            makePackageDeclaration(),
            makeImportDeclarationList(rslast),
            new RSLListDefault < TypeDeclaration > (
            makeTypeDeclaration(
            (BasicClassExpr) classExpr,
            rslast.getLibModule().
            getSchemeDef().getId()))
            )));

      }
      else {
        System.out.println("Ignoring class: " +
                           rslId2JavaName(rslast.getLibModule().
                                          getSchemeDef().
                                          getId()).getText());
        _v0 = new RSLMapDefault < String, CompilationUnit > ();
      }
    }
    return _v0;
  }

  public static RSLList < ImportDeclaration >
      makeImportDeclarationList(
      RSLAst rslast) {
```

```
    RSLList < ImportDeclaration > _v0 = null;
    if (translator.getProperty("import") != null) {
      _v0 = new RSLListDefault < ImportDeclaration > (new
          ImportDeclaration[] {
          new ImportDeclaration(
          new QualifiedName(
          new QualifiedName(
          new SimpleName("java"),
          new SimpleName("util")
          ),
          new SimpleName("*")
          )
          ),
          new ImportDeclaration(
          new QualifiedName(
          new QualifiedName(
          new SimpleName("translator"),
          new SimpleName("rsllib")
          ),
          new SimpleName("*")
          )
          )
      }).concat(makeImportDeclarationList(
          new RSLListDefault < String > (
          translator.getProperty("import").split("\\$"))));
    }
    else {
      _v0 = new RSLListDefault < ImportDeclaration > (new
          ImportDeclaration[] {
          new ImportDeclaration(
          new QualifiedName(
          new QualifiedName(
          new SimpleName("java"),
          new SimpleName("util")
          ),
          new SimpleName("*")
          )
          ),
          new ImportDeclaration(
          new QualifiedName(
          new QualifiedName(
          new SimpleName("translator"),
          new SimpleName("rsllib")
          ),
          new SimpleName("*")
          )
          )
      });
    }
```

```
    return _v0;
  }

  public static RSLList < ImportDeclaration >
     makeImportDeclarationList(
      RSLList < String > stringList) {
    RSLList < ImportDeclaration > _v0 = null;
    if (stringList.equals(new RSLListDefault < String > ())) {
      _v0 = new RSLListDefault < ImportDeclaration > ();
    }
    else {
      _v0 = (new RSLListDefault < ImportDeclaration > (
          new ImportDeclaration(
          new SimpleName(stringList.hd()))))).concat(
          makeImportDeclarationList(stringList.tl()));
    }
    return _v0;
  }

  public static OptionalPackageDeclaration
     makePackageDeclaration() {
    OptionalPackageDeclaration _v0 = null;
    if (translator.getProperty("package") != null) {
      _v0 = new PackageDeclaration(
          new SimpleName(translator.getProperty("package")));
    }
    else {
      _v0 = new NoPackageDeclaration();
    }
    return _v0;
  }

  /*
    public static RSLList<TypeDeclaration>
        makeTypeDeclarationList(
   RSLAst rslast) {
    return new RSLListDefault<TypeDeclaration>(new
        TypeDeclaration[]{
    makeTypeDeclaration(rslast.getLibModule().getSchemeDef())});
    }

    public static TypeDeclaration makeTypeDeclaration(SchemeDef
        sdef) {
    return new ClassDeclaration(
    new RSLListDefault<Modifier>(new Modifier[]{Modifier.PUBLIC
        }),
    rslId2JavaName(sdef.getId()),
    null,
    new RSLListDefault<SimpleName>(),
```

```
      new RSLListDefault<ConstructorDeclaration>(),
      makeMethodDeclarationList1(((BasicClassExpr) sdef.
          getClassExpr()).
    getDeclList()).concat(
      makeMainMethodDeclaration(((BasicClassExpr)sdef.getClassExpr
          ()).
    getDeclList())
      ),
      makeFieldDeclarationList(((BasicClassExpr) sdef.getClassExpr
          ()).
    getDeclList()),
      new RSLListDefault<ClassDeclaration>()
      //makeClassDeclarationList1(((BasicClassExpr) sdef.
          getClassExpr()).
    getDeclList())
      );
      }
    */

  public static TypeDeclaration makeTypeDeclaration(
      BasicClassExpr bce, Id id) {
    return new ClassDeclaration(
        new RSLListDefault < Modifier > (new Modifier[] {
            Modifier.PUBLIC}),
        rslId2JavaName(id),
        translator.getProperty("extendbase") != null ?
        new SimpleName(translator.getProperty("extendbase")) :
            null,
        new RSLListDefault < SimpleName > (),
        new RSLListDefault < ConstructorDeclaration > (),
        makeMethodDeclarationList1(bce.getDeclList()).concat(
        makeMainMethodDeclaration(bce.getDeclList())
        ),
        makeFieldDeclarationList(bce.getDeclList()),
        new RSLListDefault < ClassDeclaration > ()
        //makeClassDeclarationList1(((BasicClassExpr)
        sdef.getClassExpr()).getDeclList())
        );
  }

  public static RSLList < MethodDeclaration >
      makeMethodDeclarationList(
      RecordVariant rv) {
    RSLList < MethodDeclaration > _v0;
    _v0 = makeMethodDeclarationList(rv.getConstructor().getId(),
                                    rv.getComponentKindList());
    return _v0;
  }
```

```
  public static RSLList < MethodDeclaration >
      makeMethodDeclarationList(
       ShortRecordDef srd) {
335   RSLList < MethodDeclaration > _v0;
    _v0 = makeMethodDeclarationList(srd.getId(),
                                    srd.getComponentKindString()
                                        );
    return _v0;
  }
340
  public static RSLList < MethodDeclaration >
      makeMethodDeclarationList(
       Id id, RSLList < ComponentKind > componentKindList) {
    RSLList < MethodDeclaration > _v0;
    _v0 = new RSLListDefault < MethodDeclaration > (
345       new MethodDeclaration[] {
          new MethodDeclaration(
          new RSLListDefault < Modifier > (Modifier.PUBLIC),
          new SimpleName("equals"),
          PrimitiveType.JAVA_BOOLEAN,
350       new RSLListDefault < SingleVariableDeclaration > (
          new SingleVariableDeclaration(
          new RSLListDefault < Modifier > (),
          new ReferenceType(new SimpleName("Object"), null),
          new SimpleName("o"),
355       null
          )
          ),
          new Block(new RSLListDefault < Statement > (new
              Statement[] {
          new IfStatement(
360       new InstanceOfExpression(new SimpleName("o"),
                                   new ReferenceType(
                                       rslId2JavaName(id), null)),
          new Block(makeStatementListEqualsMethod(id,
              componentKindList).
                  concat(new RSLListDefault < Statement >
                          (new ReturnStatement(new BooleanLiteral
                              ("true"))))),
365       null
          ),
          new ReturnStatement(new BooleanLiteral("false"))
    }))
          ),
370       new MethodDeclaration(
          new RSLListDefault < Modifier > (Modifier.PUBLIC),
          new SimpleName("toString"),
          new ReferenceType(new SimpleName("String"), null),
          new RSLListDefault < SingleVariableDeclaration > (),
```

```
375         new Block(new RSLListDefault < Statement >
                    (new ReturnStatement(new InfixExpression(
            new StringLiteral(id.getText() + "("),
            InfixOperator.JAVA_INFIX_OP_PLUS,
            makeExpressionToStringMethod(componentKindList)
380         ))))
            )
      }
          ).concat(makeAccessorMethods(componentKindList, 0));
      return _v0;
385   }

      public static RSLList < MethodDeclaration >
          makeAccessorMethods(RSLList < ComponentKind >
              componentKindList,
                              int currentVariableNumber) {
390     RSLList < MethodDeclaration > _v0 = null;
        if (componentKindList.equals(new RSLListDefault <
            ComponentKind > ())) {
          RSLList < MethodDeclaration > _v1 = null;
          _v1 = new RSLListDefault < MethodDeclaration > ();
          _v0 = _v1;
395     }
        else {
          RSLList < MethodDeclaration > _v1 = null;
          _v1 = new RSLListDefault < MethodDeclaration > (new
              MethodDeclaration(
              new RSLListDefault < Modifier > (Modifier.PUBLIC),
400           rslId2JavaName(componentKindList.hd().
                  getOptionalDestructor().getId()),
              matchType(componentKindList.hd().getTypeExpr(), false
                  , true),
              new RSLListDefault < SingleVariableDeclaration > (),
              new Block(new RSLListDefault < Statement >
                        (new ReturnStatement(new
405                                             SimpleName("_v" +
                                                    currentVariableNumber
                                                        ))))
              )).concat(makeAccessorMethods(componentKindList.tl(),
                                            currentVariableNumber
                                                + 1));
          _v0 = _v1;
410     }
        return _v0;
      }

      public static RSLList < Statement >
415       makeStatementListEqualsMethod(Id id,
                                        RSLList < ComponentKind >
```

```
                                                componentKindList) {
    RSLList < Statement > _v0 = null;
    if (componentKindList.equals(new RSLListDefault <
        ComponentKind > ())) {
      RSLList < Statement > _v1 = null;
      _v1 = new RSLListDefault < Statement > ();
      _v0 = _v1;
    }
    else {
      RSLList < Statement > _v1 = null;
      if (matchType( (componentKindList.hd()).getTypeExpr(),
          false, false) instanceof
          PrimitiveType) {
        RSLList < Statement > _v2 = null;
        _v2 = new RSLListDefault < Statement > (
            new IfStatement(
            new PrefixExpression(
            PrefixOperator.JAVA_PREFIX_OP_NOT,
            new ParentherizedExpression(
            new InfixExpression(
            new MethodInvocation(
            new ThisExpression(null),
            rslId2JavaName( (componentKindList.hd()).
                getOptionalDestructor().
                              getId()),
            new RSLListDefault < Expression > ()
            ),
            InfixOperator.JAVA_INFIX_OP_EQUALS,
            new MethodInvocation(
            new ParentherizedExpression(
            new CastExpression(new ReferenceType(rslId2JavaName(
                id), null),
                                  new SimpleName("o"))
            ),
            rslId2JavaName( (componentKindList.hd().
                getOptionalDestructor().
                                getId())),
            new RSLListDefault < Expression > ()
            )
            )
            )
            ),
            new Block(new RSLListDefault < Statement >
                      (new ReturnStatement(new BooleanLiteral("
                          false")))),
            null
            )
            ).concat(makeStatementListEqualsMethod(id,
                componentKindList.tl()));
```

```
        _v1 = _v2;
      }
      else {
        RSLList < Statement > _v2 = null;
        _v2 = new RSLListDefault < Statement > (
            new IfStatement(
            new PrefixExpression(
            PrefixOperator.JAVA_PREFIX_OP_NOT,
            new ParentherizedExpression(
            new MethodInvocation(
            new MethodInvocation(
            new ThisExpression(null),
            rslId2JavaName( (componentKindList.hd()).
                getOptionalDestructor().
                            getId()),
            new RSLListDefault < Expression > ()
            ),
            new SimpleName("equals"),
            new RSLListDefault < Expression > (
            new MethodInvocation(
            new ParentherizedExpression(
            new CastExpression(new ReferenceType(rslId2JavaName(
                id), null),
                                new SimpleName("o"))
            ),
            rslId2JavaName(componentKindList.hd().
                getOptionalDestructor().getId()),
            new RSLListDefault < Expression > ()
            )
            )
            )
            )
            ),
            new Block(new RSLListDefault < Statement >
                      (new ReturnStatement(new BooleanLiteral("
                          false")))),
            null
            )
            ).concat(makeStatementListEqualsMethod(id,
                componentKindList.tl()));
        _v1 = _v2;
      }
      _v1 =
          _v0 = _v1;
    }
    return _v0;
  }

  public static Expression makeExpressionToStringMethod(RSLList
```

```
      < ComponentKind >
      componentKindList) {
    Expression _v0 = null;
    if (componentKindList.equals(new RSLListDefault <
        ComponentKind > ())) {
      Expression _v1 = null;
      _v1 = new StringLiteral(")");
      _v0 = _v1;
    }
    else {
      Expression _v1 = null;
      if (matchType(componentKindList.hd().getTypeExpr(), false
          , false) instanceof
          PrimitiveType) {
        Expression _v2 = null;
        if (componentKindList.tl().equals(new RSLListDefault <
            ComponentKind > ())) {
          Expression _v3 = null;
          _v3 = new InfixExpression(
              new MethodInvocation(
              new ThisExpression(null),
              rslId2JavaName( (componentKindList.hd()).
                  getOptionalDestructor().
                              getId()),
              new RSLListDefault < Expression > ()
              ),
              InfixOperator.JAVA_INFIX_OP_PLUS,
              makeExpressionToStringMethod(componentKindList.tl
                  ())
              );
          _v2 = _v3;
        }
        else {
          Expression _v3 = null;
          _v3 = new InfixExpression(
              new MethodInvocation(
              new ThisExpression(null),
              rslId2JavaName( (componentKindList.hd()).
                  getOptionalDestructor().
                              getId()),
              new RSLListDefault < Expression > ()
              ),
              InfixOperator.JAVA_INFIX_OP_PLUS,
              new InfixExpression(
              new StringLiteral(", "),
              InfixOperator.JAVA_INFIX_OP_PLUS,
              makeExpressionToStringMethod(componentKindList.tl
                  ())
              )
```

```
            );
        _v2 = _v3;
      }
      _v1 = _v2;
    }
    else {
      Expression _v2 = null;
      if (componentKindList.tl().equals(new RSLListDefault <
          ComponentKind > ())) {
        Expression _v3 = null;
        _v3 = new InfixExpression(
            new MethodInvocation(
            new MethodInvocation(
            new ThisExpression(null),
            rslId2JavaName( (componentKindList.hd()).
                getOptionalDestructor().
                            getId()),
            new RSLListDefault < Expression > ()
            ),
            new SimpleName("toString"),
            new RSLListDefault < Expression > ()
            ),
            InfixOperator.JAVA_INFIX_OP_PLUS,
            makeExpressionToStringMethod(componentKindList.tl
                ())
            );
        _v2 = _v3;
      }
      else {
        Expression _v3 = null;
        _v3 = new InfixExpression(
            new MethodInvocation(
            new MethodInvocation(
            new ThisExpression(null),
            rslId2JavaName( (componentKindList.hd()).
                getOptionalDestructor().
                            getId()),
            new RSLListDefault < Expression > ()
            ),
            new SimpleName("toString"),
            new RSLListDefault < Expression > ()
            ),
            InfixOperator.JAVA_INFIX_OP_PLUS,
            new InfixExpression(
            new StringLiteral(", "),
            InfixOperator.JAVA_INFIX_OP_PLUS,
            makeExpressionToStringMethod(componentKindList.tl
                ())
            )
```

```
                );
            _v2 = _v3;
          }
          _v1 = _v2;
        }
        _v0 = _v1;
      }
      return _v0;
    }

    public static RSLList < MethodDeclaration >
        makeMethodDeclarationList1(RSLList < Decl > dl) {
      RSLList < MethodDeclaration > _v1 = new RSLListDefault <
          MethodDeclaration > ();
      if (dl.equals(new RSLListDefault < Decl > ())) {
        _v1 = new RSLListDefault < MethodDeclaration > ();
      }
      else {
        //case hd dl of
        if (dl.hd() instanceof ValueDecl) {
          //make_ValueDecl(vdl)
          RSLList < ValueDef > vdl = ( (ValueDecl) dl.hd()).
              getValueDefList();
          _v1 = makeMethodDeclarationList2(vdl).concat(
              makeMethodDeclarationList1(
               dl.tl()));
        }
        else {
          _v1 = makeMethodDeclarationList1(dl.tl());
        }
      }
      return _v1;
    }

    public static RSLList < MethodDeclaration >
        makeMethodDeclarationList2(RSLList < ValueDef > vdl) {
      RSLList < MethodDeclaration > _v1 = new RSLListDefault <
          MethodDeclaration > ();
      if (vdl.equals(new RSLListDefault())) {
        _v1 = new RSLListDefault < MethodDeclaration > ();
      }
      else {
        //case hd vdl of
        if (vdl.hd() instanceof ExplicitFunctionDef) {
          //make_ExplicitFunctionDef(efd)
          ExplicitFunctionDef efd = (ExplicitFunctionDef) vdl.hd()
              ;
          _v1 = (new RSLListDefault < MethodDeclaration >
                (makeMethodDeclaration(efd))).concat(
```

```
                    makeMethodDeclarationList2(
                vdl.tl()));
          }
        }
        return _v1;
    }

    public static MethodDeclaration makeMethodDeclaration(
        ExplicitFunctionDef efd) {
      JavaType type = makeReturnType(efd.getSingleTyping());

      VariableDeclarationStatement vds = null;
      boolean returnValue = false;
      int currentVariableNumber = 0;
      if (type != PrimitiveType.JAVA_VOID) {
        if (type instanceof ReferenceType) {
          vds = new VariableDeclarationStatement(new
              RSLListDefault < Modifier > (),
                                                 type,
                                                 new
                                                 VariableDeclarationFragment
                                                     (new
                SimpleName("_v" + currentVariableNumber), new
                    NullLiteral()));
        }
        else if (type instanceof PrimitiveType) {
          if (type == PrimitiveType.JAVA_INT) {
            vds = new VariableDeclarationStatement(new
                RSLListDefault < Modifier > (),
                                                   type,
                                                   new
                                                   VariableDeclarationFragment
                                                       (new
                SimpleName("_v" + currentVariableNumber), new
                    IntegerLiteral("0")));
          }
          else if (type == PrimitiveType.JAVA_DOUBLE) {
            vds = new VariableDeclarationStatement(new
                RSLListDefault < Modifier > (),
                                                   type,
                                                   new
                                                   VariableDeclarationFragment
                                                       (new
                SimpleName("_v" + currentVariableNumber),
                                                   new DoubleLiteral
                                                       ("0.0d")));
          }
          else if (type == PrimitiveType.JAVA_CHAR) {
            vds = new VariableDeclarationStatement(new
```

```
                RSLListDefault < Modifier > (),
                                                    type,
                                                    new
                                                    VariableDeclarationFragment
                                                        (new
                SimpleName("_v" + currentVariableNumber), new
                    CharLiteral(" ")));
            }
            else if (type == PrimitiveType.JAVA_BOOLEAN) {
              vds = new VariableDeclarationStatement(new
                  RSLListDefault < Modifier > (),
                                                    type,
                                                    new
                                                    VariableDeclarationFragment
                                                        (new
                SimpleName("_v" + currentVariableNumber),
                                                    new BooleanLiteral
                                                        ("false")));
            }
          }
          returnValue = true;
        }
        return new MethodDeclaration(
            new RSLListDefault < Modifier >
            (new Modifier[] {Modifier.PUBLIC, Modifier.STATIC}),
            makeMethodName(efd.getFormalFunctionApplication()),
            type,
            makeArgumentList(efd.getFormalFunctionApplication(),
                             efd.getSingleTyping()),
            makeBlock(efd.getValueExpr(), vds, returnValue,
                currentVariableNumber++)
            );
    }

    public static SimpleName makeMethodName(
        FormalFunctionApplication ffa) {
      SimpleName _v1 = null;
      if (ffa instanceof IdApplication) {
        IdApplication _v2 = (IdApplication) ffa;
        Id id = _v2.getId();
        _v1 = rslId2JavaName(id);
      }
      return _v1;
    }

    public static JavaType makeReturnType(SingleTyping st) {
      JavaType _v1 = null;
      if (st.getTypeExpr() instanceof FunctionTypeExpr) {
        _v1 = matchType( ( (FunctionTypeExpr) st.getTypeExpr()).
```

```
                        getFunctionResultDescription().getTypeExpr
                            (), false, true);
    }
    return _v1;
  }

  public static RSLList < SingleVariableDeclaration >
      makeArgumentList(FormalFunctionApplication ffa,
          SingleTyping st) {
    RSLList < SingleVariableDeclaration > _v1 = null;
    TypeExpr te = st.getTypeExpr();

    if (ffa instanceof IdApplication) {
      IdApplication _v2 = (IdApplication) ffa;
      RSLList < FormalFunctionParameter >
          formalFunctionParameterList = _v2.
              getFormalFunctionParameters();

      if (te instanceof FunctionTypeExpr) {
        FunctionTypeExpr fte = (FunctionTypeExpr) te;
        TypeExpr _v3 = fte.getTypeExpr();
        if (_v3 instanceof ProductTypeExpr) {
          ProductTypeExpr _v4 = (ProductTypeExpr) _v3;
          _v1 = makeArgumentListProductTypeExpr(
              formalFunctionParameterList.hd().
                                                  getBindingList()
                                                  ,
                                                  _v4.getTypeExprList
                                                      ());
        }
        else if (_v3 instanceof TypeLiteral) {
          TypeLiteral tl = (TypeLiteral) _v3;
          if (tl != TypeLiteral.RSLUNIT) {
            SimpleName name = rslId2JavaName( (Id)
                formalFunctionParameterList.
                                                hd().getBindingList
                                                    ().hd());
            _v1 = new RSLListDefault < SingleVariableDeclaration
                > (
                new SingleVariableDeclaration(
                new RSLListDefault < Modifier > (),
                matchType(_v3, false, true),
                name,
                null
                )
                );
          }
          else {
            _v1 = new RSLListDefault < SingleVariableDeclaration
```

```
                    > ();
750           }
            }
            else if (_v3 instanceof TypeName) {
              SimpleName name = rslId2JavaName( (Id)
                  formalFunctionParameterList.hd().
                                                  getBindingList().hd()
                                                    );
755           _v1 = new RSLListDefault < SingleVariableDeclaration
                   > (
                  new SingleVariableDeclaration(
                  new RSLListDefault < Modifier > (),
                  matchType(_v3, false, true),
                  name,
760               null
                  )
                  );
            }
            else if (_v3 instanceof ListTypeExpr) {
765           if (_v3 instanceof FiniteListTypeExpr) {
                FiniteListTypeExpr _v4 = (FiniteListTypeExpr) _v3;
                SimpleName name = rslId2JavaName( (Id)
                    formalFunctionParameterList.
                                                    hd().getBindingList
                                                      ().hd());
                _v1 = new RSLListDefault < SingleVariableDeclaration
                     > (
770                 new SingleVariableDeclaration(
                    new RSLListDefault < Modifier > (),
                    matchType(_v4, false, true),
                    name,
                    null
775                 )
                    );
              }
            }
          }
        }
        return _v1;
      }

      public static RSLList < SingleVariableDeclaration >
785       makeArgumentListProductTypeExpr(RSLList < Binding >
            bindingList,
                                          RSLList < TypeExpr >
                                            exprList) {
        RSLList < SingleVariableDeclaration > _v1 = null;
        if (bindingList.equals(new RSLListDefault < Binding > ())) {
          _v1 = new RSLListDefault < SingleVariableDeclaration > ();
```

```
790       }
          else {
            SimpleName name = rslId2JavaName( (Id) bindingList.hd());
            _v1 = new RSLListDefault < SingleVariableDeclaration > (
                new SingleVariableDeclaration(
795             new RSLListDefault < Modifier > (),
                matchType(exprList.hd(), false, true),
                name,
                null
                )
800             ).concat(makeArgumentListProductTypeExpr(bindingList.
                    tl(),
                exprList.tl()));
          }
          return _v1;
        }
805
        public static RSLList < SingleVariableDeclaration >
            makeArgumentList(RSLList < ComponentKind >
                componentKindList,
                             int currentArgumentNumber) {
          RSLList < SingleVariableDeclaration > _v0 = null;
810       if (componentKindList.equals(new RSLListDefault <
              ComponentKind > ())) {
            RSLList < SingleVariableDeclaration > _v1 = null;
            _v1 = new RSLListDefault < SingleVariableDeclaration > ();
            _v0 = _v1;
          }
815       else {
            RSLList < SingleVariableDeclaration > _v1 = null;
            _v1 = (new RSLListDefault < SingleVariableDeclaration > (
                new SingleVariableDeclaration(
                new RSLListDefault < Modifier > (),
820             matchType( (componentKindList.hd()).getTypeExpr(),
                    false, true),
                new SimpleName("_v" + currentArgumentNumber),
                null
                )
                )
825                 ).concat(makeArgumentList(componentKindList.tl(),
                                              currentArgumentNumber
                                                  + 1));
            _v0 = _v1;
          }
          return _v0;
830     }

        public static Block makeBlock(ValueExpr valueExpr,
                                      VariableDeclarationStatement vds
```

```
                                    ,
                                boolean returnValue, int
                                    currentVariableNumber) {
835     Block _v1 = null;

        VariableDeclarationStatement vds2 = new
            VariableDeclarationStatement(
            new RSLListDefault < Modifier > (),
            vds.getType(),
840         new VariableDeclarationFragment(new SimpleName("_v" +
            currentVariableNumber), vds.getFragment().
                getInitialization()));

        //case valueExpr of
        if (valueExpr instanceof StructuredExpr) {
845       //make_StructuredExpr(se)
          StructuredExpr se = (StructuredExpr) valueExpr;

          if (returnValue) {
            _v1 = new Block( (new RSLListDefault < Statement > (vds)
                ).concat(
850             makeStatementList(se, vds2, currentVariableNumber).
                    concat(
                new RSLListDefault < Statement >
                (new ReturnStatement(vds.getFragment().getName()))))
                    );
          }
          else {
855         _v1 = new Block( (new RSLListDefault < Statement > (vds2
                )).concat(
                makeStatementList(se, vds2, currentVariableNumber).
                    concat(
                new RSLListDefault < Statement > (
                new ExpressionStatement(
                new AssignmentExpression(
860             vds.getFragment().getName(),
                AssignmentOperator.JAVA_ASSIGNMENT_OP_EQUAL,
                vds2.getFragment().getName()))))));
          }
        }
865     else if (valueExpr instanceof ValueInfixExpr) {
          //make_ValueInfixExpr(vie)
          ValueInfixExpr vie = (ValueInfixExpr) valueExpr;
          //_v1 = new Block(makeStatementList(vie, vds,
              currentVariableNumber));

870       if (returnValue) {
            _v1 = new Block( (new RSLListDefault < Statement > (vds)
                ).concat(
```

```
                makeStatementList(vie, vds2, currentVariableNumber
                    + 1).concat(
                new RSLListDefault < Statement >
                (new ReturnStatement(vds.getFragment().getName())))
                    );
875         }
            else {
              _v1 = new Block( (new RSLListDefault < Statement > (vds2
                  )).concat(
                makeStatementList(vie, vds2, (currentVariableNumber
                    + 1)).concat(
                new RSLListDefault < Statement > (
880             new ExpressionStatement(
                new AssignmentExpression(
                vds.getFragment().getName(),
                AssignmentOperator.JAVA_ASSIGNMENT_OP_EQUAL,
                vds2.getFragment().getName()))))));
885         }

          }
          else if (valueExpr instanceof EnumeratedListExpr) {
            EnumeratedListExpr ele = (EnumeratedListExpr) valueExpr;
890         if (returnValue) {
              _v1 = new Block( (new RSLListDefault < Statement > (vds)
                  ).concat(
                makeStatementList(ele, vds2, currentVariableNumber
                    + 1).concat(
                new RSLListDefault < Statement >
                (new ReturnStatement(vds.getFragment().getName())))
                    );
895         }
            else {
              _v1 = new Block( (new RSLListDefault < Statement > (vds2
                  )).concat(
                makeStatementList(ele, vds2, (currentVariableNumber
                    + 1)).concat(
                new RSLListDefault < Statement > (
900             new ExpressionStatement(
                new AssignmentExpression(
                vds.getFragment().getName(),
                AssignmentOperator.JAVA_ASSIGNMENT_OP_EQUAL,
                vds2.getFragment().getName()))))));
905         }
          }
          else if (valueExpr instanceof ApplicationExpr) {
            ApplicationExpr ae = (ApplicationExpr) valueExpr;

910         if (returnValue) {
              _v1 = new Block( (new RSLListDefault < Statement > (vds)
```

```
            ).concat(
             makeStatementList(ae, vds2, currentVariableNumber
                 + 1).concat(
             new RSLListDefault < Statement >
             (new ReturnStatement(vds.getFragment().getName()))))
                 );
915       }
          else {
            _v1 = new Block( (new RSLListDefault < Statement > (vds2
                )).concat(
             makeStatementList(ae, vds2, currentVariableNumber).
                 concat(
             new RSLListDefault < Statement > (
920          new ExpressionStatement(
             new AssignmentExpression(
             vds.getFragment().getName(),
             AssignmentOperator.JAVA_ASSIGNMENT_OP_EQUAL,
             vds2.getFragment().getName()))))));
925       }
        }
        else if (valueExpr instanceof ValueLiteral) {
          //make_ValueLiteral(vl)
          ValueLiteral vl = (ValueLiteral) valueExpr;
930
          if (returnValue) {
            _v1 = new Block( (new RSLListDefault < Statement > (vds)
                ).concat(
             makeStatementList(vl, vds2, currentVariableNumber
                 + 1).concat(
             new RSLListDefault < Statement >
935          (new ReturnStatement(vds.getFragment().getName()))))
                 );
          }
          else {
            _v1 = new Block( (new RSLListDefault < Statement > (vds2
                )).concat(
             makeStatementList(vl, vds2, currentVariableNumber).
                 concat(
940          new RSLListDefault < Statement > (
             new ExpressionStatement(
             new AssignmentExpression(
             vds.getFragment().getName(),
             AssignmentOperator.JAVA_ASSIGNMENT_OP_EQUAL,
945          vds2.getFragment().getName()))))));
          }
        }
        else if (valueExpr instanceof ValueOrVariableName) {
          //make_ValueLiteral(vl)
950       ValueOrVariableName vovn = (ValueOrVariableName) valueExpr
```

```
            ;

        if (returnValue) {
          _v1 = new Block( (new RSLListDefault < Statement > (vds)
              ).concat(
              makeStatementList(vovn, vds2, currentVariableNumber
                  + 1).concat(
              new RSLListDefault < Statement >
              (new ReturnStatement(vds.getFragment().getName()))))
                  );
        }
        else {
          _v1 = new Block( (new RSLListDefault < Statement > (vds2
              )).concat(
              makeStatementList(vovn, vds2, currentVariableNumber)
                  .concat(
              new RSLListDefault < Statement > (
              new ExpressionStatement(
              new AssignmentExpression(
              vds.getFragment().getName(),
              AssignmentOperator.JAVA_ASSIGNMENT_OP_EQUAL,
              vds2.getFragment().getName()))))));
        }
      }
      return _v1;
    }

    public static RSLList < Statement >
        makeStatementList(ValueExpr valueExpr,
            VariableDeclarationStatement vds,
                          int currentVariableNumber) {
      RSLList < Statement > _v1 = null;
      if (valueExpr instanceof StructuredExpr) {
        _v1 = makeStatementList( (StructuredExpr) valueExpr, vds,
                                 currentVariableNumber);
      }
      else if (valueExpr instanceof ValueInfixExpr) {
        _v1 = makeStatementList( (ValueInfixExpr) valueExpr, vds,
                                 currentVariableNumber);
      }
      else if (valueExpr instanceof EnumeratedListExpr) {
        _v1 = makeStatementList( (EnumeratedListExpr) valueExpr,
            vds,
                                 currentVariableNumber);
      }
      else if (valueExpr instanceof ValueOrVariableName) {
        _v1 = makeStatementList( (ValueOrVariableName) valueExpr,
            vds,
                                 currentVariableNumber);
```

```
    }
    else if (valueExpr instanceof ValueLiteral) {
      _v1 = makeStatementList( (ValueLiteral) valueExpr, vds,
                               currentVariableNumber);
    }
    else if (valueExpr instanceof ApplicationExpr) {
      _v1 = makeStatementList( (ApplicationExpr) valueExpr, vds,
                               currentVariableNumber);
    }
    return _v1;
  }

  public static RSLList < Statement >
      makeStatementList(StructuredExpr valueExpr,
                        VariableDeclarationStatement vds,
                        int currentVariableNumber) {
    RSLList < Statement > _v1 = null;
    if (valueExpr instanceof IfExpr) {
      _v1 = makeStatementList( (IfExpr) valueExpr, vds,
          currentVariableNumber);
    }
    else if (valueExpr instanceof CaseExpr) {
      _v1 = makeStatementList( (CaseExpr) valueExpr, vds,
          currentVariableNumber);
    }
    return _v1;
  }

  public static RSLList < Statement >
      makeStatementList(IfExpr ifExpr,
          VariableDeclarationStatement vds,
                        int currentVariableNumber) {
    ValueExpr condition = ifExpr.getCondition();
    ValueExpr valueExpr = ifExpr.getValueExpr();
    RSLList < ElsifBranch > elsifBranchList = ifExpr.
        getElsifBranchList();
    ElseBranch optionalElseBranch = ifExpr.getElseBranch();
    Block elseBlock = null;
    if (optionalElseBranch != null) {
      elseBlock = makeBlock(optionalElseBranch.getValueExpr(),
          vds, false,
                            (currentVariableNumber + 1));
    }
    Statement s = new IfStatement(makeExpression(condition),
                                  makeBlock(valueExpr, vds,
                                      false,
                                          (currentVariableNumber
                                              + 1)),
                                  elseBlock);
```

```
    return new RSLListDefault < Statement > (s);
  }

  public static RSLList < Statement >
      makeStatementList(CaseExpr caseExpr,
          VariableDeclarationStatement vds,
                        int currentVariableNumber) {
    RSLList < Statement > _v0 = null;
    Expression e = makeExpression(caseExpr.getCondition());

    _v0 = makeStatementList(caseExpr.getCaseBranchList(), e, vds
        ,
                            currentVariableNumber + 1, null);

    return _v0;
  }

  public static RSLList < Statement >
      makeStatementList(RSLList < CaseBranch > caseBranchList,
                        Expression expression,
                            VariableDeclarationStatement vds,
                        int currentVariableNumber, Statement
                            statement) {
    RSLList < Statement > _v0 = null;
    if (caseBranchList.equals(new RSLListDefault < CaseBranch
        > ())) {
      RSLList < Statement > _v1 = null;
      if (statement == null) {
        _v1 = new RSLListDefault < Statement > ();
      }
      else {
        _v1 = new RSLListDefault < Statement > (statement);
      }
      _v0 = _v1;
    }
    else {
      RSLList < Statement > _v1 = null;
      Statement s = makeStatement(caseBranchList.hd(),
          expression, vds,
                                  currentVariableNumber,
                                      statement);
      _v1 = makeStatementList(caseBranchList.tl(), expression,
          vds,
                              currentVariableNumber, s);
      _v0 = _v1;
    }
    return _v0;
  }
```

```
  public static Statement makeStatement(CaseBranch caseBranch,
1075                                    Expression expression,
                                        VariableDeclarationStatement
                                            vds,
                                        int
                                            currentVariableNumber
                                            ,
                                        Statement statement) {
    Statement _v0 = null;
1080    if (caseBranch.getPattern() instanceof ValueLiteralPattern)
            {
      ValueLiteralPattern vlp = (ValueLiteralPattern) caseBranch
          .getPattern();
      Statement _v1 = null;
      if (statement != null) {
        Statement _v2 = null;
1085      if (statement instanceof IfStatement) {
          Statement _v3 = null;
          IfStatement is = (IfStatement) statement;
          if (is.getOptionalElseBlock() == null) {
            Statement _v4 = null;
1090        _v4 = is.setOptionalElseBlock(new Block(new
                RSLListDefault<
                 Statement>(new IfStatement(
                 new InfixExpression(expression,
                                     InfixOperator.
                                         JAVA_INFIX_OP_EQUALS,
                                     makeExpression(vlp.
                                         getValueLiteral()))),
1095            makeBlock(caseBranch.getValueExpr(), vds, false,
                          currentVariableNumber),
                 null
                 ))));
            _v3 = _v4;
1100      }
          else {
            Statement _v4 = null;
            _v4 = is.setOptionalElseBlock(new Block(new
                RSLListDefault<
                 Statement>(
1105             makeStatement(caseBranch, expression, vds,
                               currentVariableNumber,
                               is.getOptionalElseBlock().
                                   getStatementList().hd())
                 )));
            _v3 = _v4;
1110      }
          _v2 = _v3;
        }
```

```
        else {
          //System.out.println("Somethings really wrong!!");
        }
        _v1 = _v2;
      }
      else {
        Statement _v2 = null;
        _v2 = new IfStatement(
            new InfixExpression(expression, InfixOperator.
                JAVA_INFIX_OP_EQUALS,
                                makeExpression(vlp.
                                    getValueLiteral())),
            makeBlock(caseBranch.getValueExpr(), vds, false,
                      currentVariableNumber),
            null
            );
        _v1 = _v2;
      }
      _v0 = _v1;
    }
    else if (caseBranch.getPattern() instanceof NamePattern) {
      NamePattern np = (NamePattern) caseBranch.getPattern();
      Statement _v1 = null;
      if (statement != null) {
        Statement _v2 = null;
        if (statement instanceof IfStatement) {
          Statement _v3 = null;
          IfStatement is = (IfStatement) statement;
          if (is.getOptionalElseBlock() == null) {
            Statement _v4 = null;
            _v4 = is.setOptionalElseBlock(new Block(new
                RSLListDefault<
                 Statement>(
                 new IfStatement(
                 new InstanceOfExpression(expression,
                                          new ReferenceType(
                                              rslId2JavaName(np),
                                              null)),
                 makeBlock(caseBranch.getValueExpr(), vds, false,
                           currentVariableNumber),
                 null
                 ))));
            _v3 = _v4;
          }
          else {
            Statement _v4 = null;
            _v4 = is.setOptionalElseBlock(new Block(new
                RSLListDefault<
                 Statement>(
```

```
                makeStatement(caseBranch, expression, vds,
                              currentVariableNumber,
                              is.getOptionalElseBlock().
                                  getStatementList().hd())
                ))));
1160            _v3 = _v4;
              }
              _v2 = _v3;
            }
            else {
1165          //System.out.println("Somethings really wrong!!");
            }
            _v1 = _v2;
          }
          else {
1170        Statement _v2 = null;
            _v2 = new IfStatement(
                new InstanceOfExpression(expression,
                                         new ReferenceType(
                                             rslId2JavaName(np), null
                                             )),
                makeBlock(caseBranch.getValueExpr(), vds, false,
1175                      currentVariableNumber),
                null
                );

            _v1 = _v2;
          }
1180      _v0 = _v1;
        }
        else if (caseBranch.getPattern() instanceof RecordPattern) {
          RecordPattern rp = (RecordPattern) caseBranch.getPattern()
              ;
1185      Statement _v1 = null;
          if (statement != null) {
            Statement _v2 = null;
            if (statement instanceof IfStatement) {
              Statement _v3 = null;
1190          IfStatement is = (IfStatement) statement;
              if (is.getOptionalElseBlock() == null) {
                Statement _v4 = null;
                _v4 = is.setOptionalElseBlock(new Block(new
                    RSLListDefault<
                     Statement>(
1195                 new IfStatement(
                     new InstanceOfExpression(expression,
                                              new ReferenceType(
                                                  rslId2JavaName(rp),
                                                  null)),
```

```
                new Block(
                makeStatementList(expression, rp, rp.
                    getInnerPatternList(), vds,
                                currentVariableNumber).concat(
                makeStatementList(caseBranch.getValueExpr(), vds
                    ,
                                currentVariableNumber)
                )
                ),
                null
                )
                )));
              _v3 = _v4;
            }
            else {
              Statement _v4 = null;
              _v4 = is.setOptionalElseBlock(new Block(new
                  RSLListDefault<
                  Statement>(
                  makeStatement(caseBranch, expression, vds,
                                currentVariableNumber,
                                is.getOptionalElseBlock().
                                    getStatementList().hd())
                  )));
              _v3 = _v4;
            }
            _v2 = _v3;
          }
          else {
            //System.out.println("Somethings really wrong!!");
          }
          _v1 = _v2;
        }
        else {
          Statement _v2 = null;
          _v2 = new IfStatement(
              new InstanceOfExpression(expression,
                                       new ReferenceType(
                                           rslId2JavaName(rp), null
                                           )),
              new Block(
              makeStatementList(expression, rp, rp.
                  getInnerPatternList(), vds,
                                currentVariableNumber).concat(
              makeStatementList(caseBranch.getValueExpr(), vds,
                                currentVariableNumber)
              )
              ),
              null
```

```
1240              );
            _v1 = _v2;
          }
          _v0 = _v1;
        }
1245    else if (caseBranch.getPattern() instanceof WildcardPattern)
            {
          Statement _v1 = null;
          if (statement != null) {
            Statement _v2 = null;
            if (statement instanceof IfStatement) {
1250          Statement _v3 = null;
              IfStatement is = (IfStatement) statement;
              if (is.getOptionalElseBlock() == null) {
                Statement _v4 = null;
                _v4 = ( (IfStatement) statement).
                    setOptionalElseBlock(makeBlock(
1255                 caseBranch.getValueExpr(), vds, false,
                         currentVariableNumber));
                _v3 = _v4;
              }
              else {
                Statement _v4 = null;
1260            _v4 = is.setOptionalElseBlock(new Block(new
                    RSLListDefault<
                     Statement>(
                     makeStatement(caseBranch, expression, vds,
                                   currentVariableNumber,
                                   is.getOptionalElseBlock().
                                       getStatementList().hd())
1265                 )));
                _v3 = _v4;
              }
              _v2 = _v3;
            }
1270        else {
              //System.out.println("Something is really wrong!!!");
            }
            _v1 = _v2;
          }
1275      else {
            Statement _v2 = null;
            _v2 = new ExpressionStatement(makeExpression(caseBranch.
                getValueExpr()));
            _v1 = _v2;
          }
1280      _v0 = _v1;
        }
        return _v0;
```

```
  }

1285  public static RSLList < Statement >
        makeStatementList(Expression expression, RecordPattern
            recordPattern,
                          RSLList < Pattern > patternList,
                          VariableDeclarationStatement vds,
                          int currentVariableNumber) {
1290    RSLList < Statement > _v0 = null;
        if (patternList.equals(new RSLListDefault < Pattern > ())) {
          RSLList < Statement > _v1 = null;
          _v1 = new RSLListDefault < Statement > ();
          _v0 = _v1;
1295    }
        else {
          RSLList < Statement > _v1 = null;
          _v1 = new RSLListDefault < Statement > (
              new VariableDeclarationStatement(
1300          new RSLListDefault < Modifier > (),
              matchType( ( (NamePattern) patternList.hd()).
                        getTypeEvaluatorForTypeEvaluation(), false,
                            true),
              new VariableDeclarationFragment(
              rslId2JavaName( (NamePattern) patternList.hd()),
1305          new MethodInvocation(
              new ParentherizedExpression(
              new CastExpression(
              new ReferenceType(
              rslId2JavaName(recordPattern.getValueOrVariableName().
                  getId()),
1310          null
              ),
              expression
              )
              ),
1315          rslId2JavaName( ( (NamePattern) patternList.hd()).
                  getValueInitializer()),
              new RSLListDefault < Expression > ())
              )
              ).concat(makeStatementList(expression, recordPattern,
                  patternList.tl(),
1320                                     vds, currentVariableNumber)
                                             );
          _v0 = _v1;
        }
        return _v0;
      }
1325
```

```
  public static RSLList < Statement >
      makeStatementList(ValueInfixExpr vie,
          VariableDeclarationStatement vds,
                        int currentVariableNumber) {
    RSLList < Statement > _v1 = null;

    Expression expression = makeExpression(vie);

    if (vds != null) {
      _v1 = new RSLListDefault < Statement > (
          new ExpressionStatement(
          new AssignmentExpression(
          vds.getFragment().getName(),
          AssignmentOperator.JAVA_ASSIGNMENT_OP_EQUAL,
          expression
          )
          )
          );
    }
    else {
      _v1 = new RSLListDefault < Statement > (
          new ExpressionStatement(expression)
          );
    }
    return _v1;
  }

  public static RSLList < Statement >
      makeStatementList(EnumeratedListExpr ele,
                        VariableDeclarationStatement vds,
                        int currentVariableNumber) {
    RSLList < Statement > _v1 = null;
    if (vds != null) {
      _v1 = new RSLListDefault < Statement > (
          new ExpressionStatement(
          new AssignmentExpression(
          vds.getFragment().getName(),
          AssignmentOperator.JAVA_ASSIGNMENT_OP_EQUAL,
          new ClassInstanceCreation(
          null,
          (ReferenceType) matchType(ele.
              getTypeEvaluatorForTypeEvaluation(), true, false),
          makeExpressionList(ele.getValueExprList())
          )
          )
          )
          );
    }
    else {
```

```
        _v1 = new RSLListDefault < Statement > (
            new ExpressionStatement(
            new ClassInstanceCreation(
            null,
            (ReferenceType) matchType(ele.
                getTypeEvaluatorForTypeEvaluation(), true, false),
            makeExpressionList(ele.getValueExprList())
            )
            )
            );
      }
      return _v1;
    }

    public static RSLList < Expression >
        makeExpressionList(RSLList < ValueExpr > vel) {
      RSLList < Expression > _v1 = null;
      if (vel.equals(new RSLListDefault < Expression > ())) {
        _v1 = new RSLListDefault < Expression > ();
      }
      else {
        _v1 = new RSLListDefault < Expression >
            (makeExpression(vel.hd())).concat(makeExpressionList(
                vel.tl()));
      }
      return _v1;
    }

    public static RSLList < Statement >
        makeStatementList(ApplicationExpr ae,
            VariableDeclarationStatement vds,
                        int currentVariableNumber) {
      RSLList < Statement > _v1 = null;
      if (vds != null) {
        _v1 = new RSLListDefault < Statement >
            (new ExpressionStatement(new
                                    AssignmentExpression(vds.
                                        getFragment().getName(),
            AssignmentOperator.JAVA_ASSIGNMENT_OP_EQUAL,
                makeExpression(ae))));
      }
      else {
        _v1 = new RSLListDefault < Statement >
            (new ExpressionStatement(makeExpression(ae)));
      }
      return _v1;
    }

    public static RSLList < Statement >
```

```
      makeStatementList(ValueLiteral vl,
          VariableDeclarationStatement vds,
                        int currentVariableNumber) {
    RSLList < Statement > _v1 = null;
    //case vl of
    if (vl instanceof ValueLiteralInteger || vl instanceof
        ValueLiteralReal ||
        vl instanceof ValueLiteralText || vl instanceof
            ValueLiteralChar ||
        vl instanceof ValueLiteralBool) {
      if (vds != null) {
        _v1 = new RSLListDefault < Statement >
            (new ExpressionStatement(new
                                     AssignmentExpression(vds.
                                         getFragment().getName(),
            AssignmentOperator.JAVA_ASSIGNMENT_OP_EQUAL,
                makeExpression(vl))));
      }
      else {
        _v1 = new RSLListDefault < Statement >
            (new ExpressionStatement(makeExpression(vl)));
      }
    }
    return _v1;
  }

  public static RSLList < Statement >
      makeStatementList(ValueOrVariableName vovn,
                        VariableDeclarationStatement vds,
                        int currentVariableNumber) {
    RSLList < Statement > _v0 = null;
    if (vds != null) {
      _v0 = new RSLListDefault < Statement >
          (new ExpressionStatement(new
                                   AssignmentExpression(vds.
                                       getFragment().getName(),
          AssignmentOperator.JAVA_ASSIGNMENT_OP_EQUAL,
              makeExpression(vovn))));
    }
    else {
      _v0 = new RSLListDefault < Statement >
          (new ExpressionStatement(makeExpression(vovn)));
    }
    return _v0;

  }

  public static Expression makeExpression(ValueExpr valueExpr) {
    Expression _v1 = null;
```

```
    if (valueExpr instanceof ValueInfixExpr) {
1460  Expression _v2 = null;
      ValueInfixExpr vie = (ValueInfixExpr) valueExpr;
      if (vie.getOp() == RSLInfixOp.RSL_INFIX_OP_EQUALS) {
        if (matchType(vie.getLeft().
            getTypeEvaluatorForTypeEvaluation(), false, false)
            instanceof
            ReferenceType) {
1465      _v2 = new MethodInvocation(
              makeExpression(vie.getLeft()),
              new SimpleName("equals"),
              new RSLListDefault < Expression > (makeExpression(
                  vie.getRight()))
              );
1470    }
        else {
          _v2 = new InfixExpression(
              makeExpression(vie.getLeft()),
              InfixOperator.JAVA_INFIX_OP_EQUALS,
1475          makeExpression(vie.getRight())
              );
        }
      }
      else if (vie.getOp() == RSLInfixOp.RSL_INFIX_OP_PLUS) {
1480    _v2 = new InfixExpression(
            makeExpression(vie.getLeft()),
            InfixOperator.JAVA_INFIX_OP_PLUS,
            makeExpression(vie.getRight())
            );
1485  }
      else if (vie.getOp() == RSLInfixOp.RSL_INFIX_OP_STAR) {
        _v2 = new InfixExpression(
            makeExpression(vie.getLeft()),
            InfixOperator.JAVA_INFIX_OP_STAR,
1490        makeExpression(vie.getRight())
            );
      }
      else if (vie.getOp() == RSLInfixOp.RSL_INFIX_OP_HAT) {
        _v2 = new MethodInvocation(
1495        makeExpression(vie.getLeft()),
            new SimpleName("concat"),
            new RSLListDefault < Expression > (makeExpression(
                vie.getRight()))
            );
      }
1500  _v1 = _v2;
    }
    else if (valueExpr instanceof ValuePrefixExpr) {
      Expression _v2 = null;
```

```
        ValuePrefixExpr vpe = (ValuePrefixExpr) valueExpr;
1505    if (vpe.getOp() == RSLPrefixOp.RSL_PREFIX_OP_HD) {
          _v2 = new MethodInvocation(makeExpression(vpe.
              getValueExpr()),
                                     new SimpleName("hd"),
                                     new RSLListDefault <
                                         Expression > ());
        }
1510    else if (vpe.getOp() == RSLPrefixOp.RSL_PREFIX_OP_TL) {
          _v2 = new MethodInvocation(makeExpression(vpe.
              getValueExpr()),
                                     new SimpleName("tl"),
                                     new RSLListDefault <
                                         Expression > ());
        }
1515    _v1 = _v2;
      }
      else if (valueExpr instanceof ApplicationExpr) {
        ApplicationExpr ae = (ApplicationExpr) valueExpr;
        //System.out.println("ApplicationExpr: " + ae.getValueExpr
            () + " TypeEvaluator: " + ae.
            getTypeEvaluatorForTypeEvaluation());
1520    if (ae.getTypeEvaluatorForTypeEvaluation() instanceof
            Constructor) {
          _v1 = new ClassInstanceCreation(null,
                                          new ReferenceType(new
                                              SimpleName( ( (
                                              Constructor)
              ae.getTypeEvaluatorForTypeEvaluation()).getId().
                  getText()), null),
                                          makeMethodArgumentList(
                                              ae.
1525              getOptionalValueExprList()));
        }
        else if (ae.getTypeEvaluatorForTypeEvaluation() instanceof
            ShortRecordDef) {
          _v1 = new ClassInstanceCreation(null,
                                          new ReferenceType(new
                                              SimpleName( ( (
                                              ShortRecordDef)
1530          ae.getTypeEvaluatorForTypeEvaluation()).getId().
                  getText()), null),
                                          makeMethodArgumentList(
                                              ae.
              getOptionalValueExprList()));
        }
        else if (ae.getTypeEvaluatorForTypeEvaluation() instanceof
            TypeName) {
1535      _v1 = new ClassInstanceCreation(null,
```

```
                                                    new ReferenceType(new
                                                        SimpleName( ( (
                                                        TypeName)
                ae.getTypeEvaluatorForTypeEvaluation()).getId().
                    getText()), null),
                                                    makeMethodArgumentList(
                                                        ae.
                getOptionalValueExprList()));
1540        }
            else if (ae.getTypeEvaluatorForTypeEvaluation() instanceof
                Destructor) {
              _v1 = new MethodInvocation(new ParentherizedExpression(
                  makeExpression(
                  ae.getOptionalValueExprList().hd())),
                                         rslId2JavaName( ( (Destructor
                                             ) ae.
1545                getTypeEvaluatorForTypeEvaluation()).getId()),
                                         new RSLListDefault <
                                             Expression > ());
            }
            else {
              _v1 = new MethodInvocation(null, makeMethodName(ae.
                  getValueExpr()),
1550                                     makeMethodArgumentList(ae.
                  getOptionalValueExprList()));
            }

          }
1555      else if (valueExpr instanceof ListExpr) {
            ListExpr le = (ListExpr) valueExpr;
            Expression _v2 = null;
            if (valueExpr instanceof EnumeratedListExpr) {
              EnumeratedListExpr ele = (EnumeratedListExpr) le;
1560          Expression _v3 = null;
              if (ele.getValueExprList().equals(new RSLListDefault <
                  ValueExpr > ())) {
                Expression _v4 = null;
                _v4 = new ClassInstanceCreation(
                    null,
1565                /* new ReferenceType(new SimpleName("
                        RSLListDefault"), new ReferenceType(new
                        SimpleName("Integer"), null)),*/
                    /*new ReferenceType(new SimpleName("RSLListDefault
                        "), null),*/
                    (ReferenceType) matchType(ele.
                        getTypeEvaluatorForTypeEvaluation(), true,
                        false),
                    new RSLListDefault < Expression > ()
                    );
```

```
1570            _v3 = _v4;
              }
              else {
                Expression _v4 = null;
                _v4 = new ClassInstanceCreation(
1575                null,
                    (ReferenceType)
                    matchType(ele.getTypeEvaluatorForTypeEvaluation()
                        , true, false),
                    new RSLListDefault < Expression > (
                    new ArrayCreation(matchType( ( (FiniteListTypeExpr
                        ) ele.
1580                                                  getTypeEvaluatorForTypeEvaluation
                                                        ()).
                                                    getTypeExpr(), true,
                                                      false),
                                      null,
                                      makeMethodArgumentList(ele.
                                          getValueExprList())
                                      )
1585                )
                    );
                _v3 = _v4;
              }
              _v2 = _v3;
1590        }
            _v1 = _v2;
          }
          else if (valueExpr instanceof ValueOrVariableName) {
            ValueOrVariableName vovn = (ValueOrVariableName) valueExpr
                ;
1595        if (vovn.getTypeEvaluatorForTypeEvaluation() instanceof
                Constructor) {
              _v1 = new ClassInstanceCreation(null,
                                              (ReferenceType)
                                                  matchType(vovn.
                  getTypeEvaluatorForTypeEvaluation(), true, false),
                                              new RSLListDefault <
                                                  Expression > ()
1600                                          );
            }
            else {
              _v1 = new SimpleName(vovn.getId().getText());
            }
1605      }
          else if (valueExpr instanceof ValueLiteral) {
            Expression _v2 = null;
            if (valueExpr instanceof ValueLiteralBool) {
              ValueLiteralBool vlb = (ValueLiteralBool) valueExpr;
```

```
1610          _v2 = new BooleanLiteral(vlb.getText());
            }
            else if (valueExpr instanceof ValueLiteralInteger) {
              ValueLiteralInteger vli = (ValueLiteralInteger)
                  valueExpr;
              _v2 = new IntegerLiteral(vli.getText());
1615        }
            else if (valueExpr instanceof ValueLiteralReal) {
              ValueLiteralReal vlr = (ValueLiteralReal) valueExpr;
              _v2 = new DoubleLiteral(vlr.getText());
            }
1620        else if (valueExpr instanceof ValueLiteralChar) {
              ValueLiteralChar vlc = (ValueLiteralChar) valueExpr;
              _v2 = new CharLiteral(vlc.getText());
            }
            else if (valueExpr instanceof ValueLiteralText) {
1625          ValueLiteralText vlt = (ValueLiteralText) valueExpr;
              _v2 = new StringLiteral(vlt.getText());
            }
            _v1 = _v2;
          }
1630      else {
            throw new RuntimeException("Not translated expression: "
                + valueExpr);
          }
          return _v1;
        }
1635
        public static SimpleName makeMethodName(ValueExpr ve) {
          SimpleName _v1 = null;
          if (ve instanceof ValueOrVariableName) {
            SimpleName _v2 = null;
1640        ValueOrVariableName vovn = (ValueOrVariableName) ve;
            _v2 = new SimpleName(vovn.getId().getText());
            _v1 = _v2;
          }
          return _v1;
1645    }

        public static RSLList < Expression >
            makeMethodArgumentList(RSLList < ValueExpr > rslList) {
          RSLList < Expression > _v1 = null;
1650      if (rslList == null || rslList.equals(new RSLListDefault <
              Expression > ())) {
            _v1 = new RSLListDefault < Expression > ();
          }
          else {
            _v1 = new RSLListDefault < Expression >
1655            (makeExpression(rslList.hd())).
```

```
            concat(makeMethodArgumentList(rslList.tl()));
        }
        return _v1;
    }

    public static RSLList<FieldDeclaration>
        makeFieldDeclarationList(RSLList<Decl> dl) {
        return new RSLListDefault<FieldDeclaration>();
    }

    public static RSLList<FieldDeclaration>
        makeFieldDeclarationList(RecordVariant rv, int
            currentArgumentNumber) {
        RSLList<FieldDeclaration> _v0 = null;
        _v0 = makeFieldDeclarationList1(rv.getComponentKindList(),
                                        currentArgumentNumber);
        return _v0;
    }

    public static RSLList<FieldDeclaration>
        makeFieldDeclarationList(ShortRecordDef srd, int
            currentArgumentNumber) {
        RSLList<FieldDeclaration> _v0 = null;
        _v0 = makeFieldDeclarationList1(srd.getComponentKindString()
            ,
                                        currentArgumentNumber);
        return _v0;
    }

    public static RSLList<FieldDeclaration>
        makeFieldDeclarationList1(RSLList<ComponentKind>
            componentKindList,
                                  int currentArgumentNumber) {
        RSLList<FieldDeclaration> _v0 = null;
        if (componentKindList.equals(new RSLListDefault<
            ComponentKind>())) {
            RSLList<FieldDeclaration> _v1 = null;
            _v1 = new RSLListDefault<FieldDeclaration>();
            _v0 = _v1;
        }
        else {
            RSLList<FieldDeclaration> _v1 = null;
            _v1 = (new RSLListDefault<FieldDeclaration>(
                new FieldDeclaration(
                new RSLListDefault<Modifier>(Modifier.PRIVATE),
                matchType((componentKindList.hd()).getTypeExpr(),
                    false, true),
                new VariableDeclarationFragment(new SimpleName("_v" +
                currentArgumentNumber), null)
```

```
            )
            )).concat(makeFieldDeclarationList1(componentKindList.
                tl(),
                                                currentArgumentNumber
                                                    + 1));
      _v0 = _v1;
    }
    return _v0;
  }

  public static RSLList < ClassDeclaration >
      makeClassDeclarationList1(RSLList < Decl > dl) {
    RSLList < ClassDeclaration > _v0 = null;
    if (dl.equals(new RSLListDefault < Decl > ())) {
      RSLList < ClassDeclaration > _v1 = null;
      _v1 = new RSLListDefault < ClassDeclaration > ();
      _v0 = _v1;
    }
    else {
      if (dl.hd() instanceof TypeDecl) {
        RSLList < TypeDef > tdl = ( (TypeDecl) dl.hd()).
            getTypeDefList();
        RSLList < ClassDeclaration > _v1 = null;
        _v1 = makeClassDeclarationList2(tdl).concat(
            makeClassDeclarationList1(
             dl.tl()));
        _v0 = _v1;
      }
      else {
        RSLList < ClassDeclaration > _v1 = null;
        _v1 = makeClassDeclarationList1(dl.tl());
        _v0 = _v1;
      }
    }
    return _v0;
  }

  public static RSLList < ClassDeclaration >
      makeClassDeclarationList2(RSLList < TypeDef > tdl) {
    RSLList < ClassDeclaration > _v0 = null;

    if (tdl.equals(new RSLListDefault < TypeDef > ())) {
      RSLList < ClassDeclaration > _v1 = null;
      _v1 = new RSLListDefault < ClassDeclaration > ();
      _v0 = _v1;
    }
    else {
      //System.out.println("tdl.hd(): " + tdl.hd());
      if (tdl.hd() instanceof SortDef) {
```

```
        SortDef sd = (SortDef) tdl.hd();
1745    RSLList < ClassDeclaration > _v1 = null;
        _v1 = (new RSLListDefault < ClassDeclaration > (
            makeClassDeclaration(sd))).
             concat(makeClassDeclarationList2(tdl.tl()));
        _v0 = _v1;
      }
1750  else if (tdl.hd() instanceof VariantDef) {
        VariantDef vd = (VariantDef) tdl.hd();
        RSLList < ClassDeclaration > _v1 = null;
        _v1 = (new RSLListDefault < ClassDeclaration >
               (makeClassDeclarationList(vd))).concat(
                   makeClassDeclarationList2(
1755           tdl.tl()));
        _v0 = _v1;
      }
      else if (tdl.hd() instanceof ShortRecordDef) {
        ShortRecordDef srd = (ShortRecordDef) tdl.hd();
1760    RSLList < ClassDeclaration > _v1 = null;
        _v1 = (new RSLListDefault < ClassDeclaration >
               (makeClassDeclaration(srd))).concat(
                   makeClassDeclarationList2(
           tdl.tl()));
        _v0 = _v1;
1765  }
    }
    return _v0;
  }

1770 public static ClassDeclaration makeClassDeclaration(SortDef sd
     ) {
    return new ClassDeclaration(
        new RSLListDefault < Modifier > (new Modifier[] {
            Modifier.PUBLIC}),
        rslId2JavaName(sd.getId()),
        translator.getProperty("extend") != null ?
1775    new SimpleName(translator.getProperty("extend")) : null,
        new RSLListDefault < SimpleName > (),
        new RSLListDefault < ConstructorDeclaration > (),
        new RSLListDefault < MethodDeclaration > (),
        new RSLListDefault < FieldDeclaration > (),
1780    new RSLListDefault < ClassDeclaration > ()
        );
  }

  public static RSLList < ClassDeclaration >
1785     makeClassDeclarationList(VariantDef vd) {
    RSLList < ClassDeclaration > _v0 = null;
    _v0 = new RSLListDefault < ClassDeclaration >
```

```
        (makeClassDeclaration(vd.getId())).concat(
            makeClassDeclarationList(vd.
        getVariantList(), vd.getId()));
1790    return _v0;
    }

    public static RSLList<ClassDeclaration>
        makeClassDeclarationList(RSLList<Variant> vl, Id id) {
1795    RSLList<ClassDeclaration> _v0 = null;
    if (vl.equals(new RSLListDefault<Variant>())) {
      RSLList<ClassDeclaration> _v1 = null;
      _v1 = new RSLListDefault<ClassDeclaration>();
      _v0 = _v1;
1800  }
    else {
      RSLList<ClassDeclaration> _v1 = null;
      _v1 = new RSLListDefault<ClassDeclaration>
          (makeClassDeclaration(vl.hd(),
1805      id)).concat(makeClassDeclarationList(vl.tl(), id));
      _v0 = _v1;
    }
    return _v0;
    }
1810
    public static ClassDeclaration makeClassDeclaration(Variant v
        , Id id) {
    ClassDeclaration _v0 = null;
    if (v instanceof Constructor) {
      Constructor c = (Constructor) v;
1815  ClassDeclaration _v1 = null;
      _v1 = new ClassDeclaration(
          new RSLListDefault<Modifier>(new Modifier[] {
              Modifier.PUBLIC}),
          rslId2JavaName(c.getId()),
          rslId2JavaName(id),
1820      new RSLListDefault<SimpleName>(),
          new RSLListDefault<ConstructorDeclaration>(
          new ConstructorDeclaration(
          new RSLListDefault<Modifier>(Modifier.PUBLIC),
          rslId2JavaName(c.getId()),
1825      new RSLListDefault<SingleVariableDeclaration>(),
          new Block(new RSLListDefault<Statement>())
          )
          ),
          new RSLListDefault<MethodDeclaration>(new
              MethodDeclaration[] {
1830      new MethodDeclaration(
          new RSLListDefault<Modifier>(Modifier.PUBLIC),
          new SimpleName("equals"),
```

```
            PrimitiveType.JAVA_BOOLEAN,
            new RSLListDefault < SingleVariableDeclaration > (
1835        new SingleVariableDeclaration(
            new RSLListDefault < Modifier > (),
            new ReferenceType(new SimpleName("Object"), null),
            new SimpleName("o"),
            null
1840        )
            ),
            new Block(new RSLListDefault < Statement > (new
                Statement[] {
            new IfStatement(
            new InstanceOfExpression(new SimpleName("o"),
1845                                 new ReferenceType(
                                         rslId2JavaName(c.getId())
                                         , null)),
            new Block(new RSLListDefault < Statement >
                      (new ReturnStatement(new BooleanLiteral("
                          true")))),
            null
            ),
1850                     new ReturnStatement(new BooleanLiteral("
                             false"))
        }))
            ),
            new MethodDeclaration(
            new RSLListDefault < Modifier > (Modifier.PUBLIC),
1855        new SimpleName("toString"),
            new ReferenceType(new SimpleName("String"), null),
            new RSLListDefault < SingleVariableDeclaration > (),
            new Block(new RSLListDefault < Statement >
                      (new ReturnStatement(new StringLiteral(c.
                          getId().getText()))))
1860        )
        }).concat(makeVisitorMethod(c.getId(), false)),
            new RSLListDefault < FieldDeclaration > (),
            new RSLListDefault < ClassDeclaration > ()
            );
1865      _v0 = _v1;
      }
      else if (v instanceof RecordVariant) {
        RecordVariant rv = (RecordVariant) v;
        ClassDeclaration _v1 = null;
1870    _v1 = new ClassDeclaration(
            new RSLListDefault < Modifier > (new Modifier[] {
                Modifier.PUBLIC}),
            rslId2JavaName(rv.getConstructor().getId()),
            rslId2JavaName(id),
            new RSLListDefault < SimpleName > (),
```

```
1875            new RSLListDefault < ConstructorDeclaration > (
                new ConstructorDeclaration(
                new RSLListDefault < Modifier > (Modifier.PUBLIC),
                rslId2JavaName(rv.getConstructor().getId()),
                makeArgumentList(rv.getComponentKindList(), 0),
1880            new Block(makeConstructorStatementList(rv.
                    getComponentKindList(), 0))
                )
                ),
                makeMethodDeclarationList(rv).concat(makeVisitorMethod
                    (rv.
                getConstructor().getId(), false)),
1885            makeFieldDeclarationList(rv, 0),
                new RSLListDefault < ClassDeclaration > ()
                );
          _v0 = _v1;
        }
1890    return _v0;
    }

    public static ClassDeclaration makeClassDeclaration(Id id) {
      return new ClassDeclaration(
1895        new RSLListDefault < Modifier >
            (new Modifier[] {Modifier.PUBLIC, Modifier.ABSTRACT}),
            rslId2JavaName(id),
            translator.getProperty("extend") != null ?
            new SimpleName(translator.getProperty("extend")) : null,
1900        new RSLListDefault < SimpleName > (),
            new RSLListDefault < ConstructorDeclaration > (),
            new RSLListDefault < MethodDeclaration > (new
                MethodDeclaration[] {
                                                    new MethodDeclaration
                                                        (
            new RSLListDefault < Modifier >
1905        (new Modifier[] {Modifier.PUBLIC, Modifier.ABSTRACT}),
            new SimpleName("equals"),
            PrimitiveType.JAVA_BOOLEAN,
            new RSLListDefault < SingleVariableDeclaration > (
            new SingleVariableDeclaration(new RSLListDefault <
                Modifier > (),
1910                                        new ReferenceType(new
                                                SimpleName("Object"),
                                                null),
                                            new SimpleName("o"),
                                            null)),
            null
            ),
1915                    new MethodDeclaration(
            new RSLListDefault < Modifier >
```

```
        (new Modifier[] {Modifier.PUBLIC, Modifier.ABSTRACT}),
        new SimpleName("toString"),
        new ReferenceType(new SimpleName("String"), null),
        new RSLListDefault < SingleVariableDeclaration > (),
        null
        ),

    }).concat(makeVisitorMethod(id, true)),
        new RSLListDefault < FieldDeclaration > (),
        new RSLListDefault < ClassDeclaration > ()
        );
  }

  public static ClassDeclaration makeClassDeclaration(
      ShortRecordDef srd) {
    return new ClassDeclaration(
        new RSLListDefault < Modifier > (new Modifier[] {
            Modifier.PUBLIC}),
        rslId2JavaName(srd.getId()),
        translator.getProperty("extend") != null ?
        new SimpleName(translator.getProperty("extend")) : null,
        new RSLListDefault < SimpleName > (),
        new RSLListDefault < ConstructorDeclaration > (
        new ConstructorDeclaration(
        new RSLListDefault < Modifier > (Modifier.PUBLIC),
        rslId2JavaName(srd.getId()),
        makeArgumentList(srd.getComponentKindString(), 0),
        new Block(makeConstructorStatementList(srd.
            getComponentKindString(), 0))
        )
        ),
        makeMethodDeclarationList(srd).concat(makeVisitorMethod(
            srd.getId(), false)),
        makeFieldDeclarationList(srd, 0),
        new RSLListDefault < ClassDeclaration > ()
        );
  }

  public static RSLList < Statement >
      makeConstructorStatementList(RSLList < ComponentKind >
          componentKindList,
                                   int currentArgumentNumber) {
    RSLList < Statement > _v0 = null;
    if (componentKindList.equals(new RSLListDefault < Statement
        > ())) {
      RSLList < Statement > _v1 = null;
      _v1 = new RSLListDefault < Statement > ();
      _v0 = _v1;
    }
```

```
1960      else {
            RSLList < Statement > _v1 = null;
            _v1 = new RSLListDefault < Statement > (new
                ExpressionStatement(
                new AssignmentExpression(
                new FieldAccessExpression(new ThisExpression(null),
1965                                      new SimpleName("_v" +
                                              currentArgumentNumber)),
                AssignmentOperator.JAVA_ASSIGNMENT_OP_EQUAL,
                //rslId2JavaName(componentKindList.hd().
                    getOptionalDestructor())
                new SimpleName("_v" + currentArgumentNumber)
                )
1970            )).concat(makeConstructorStatementList(
                    componentKindList.tl(),
                                                       currentArgumentNumber
                                                           + 1));
            _v0 = _v1;
          }
          return _v0;
1975    }

        public static RSLList < MethodDeclaration >
            makeVisitorMethod(Id id, boolean abstractMethod) {
          RSLList < MethodDeclaration > _v0 = null;
1980      if (createVisitorMethods) {
            if (abstractMethod) {
              _v0 = new RSLListDefault < MethodDeclaration > (
                  new MethodDeclaration(
                  new RSLListDefault < Modifier >
1985              (new Modifier[] {Modifier.PUBLIC, Modifier.ABSTRACT
                      }),
                  new SimpleName("accept"),
                  PrimitiveType.JAVA_VOID,
                  new RSLListDefault < SingleVariableDeclaration >
                  (new SingleVariableDeclaration(new RSLListDefault <
                      Modifier > (),
1990                                           new ReferenceType( (
                                                   translator.
                  getProperty("visitor") != null ?
                                               new SimpleName(
                                                   translator.
                  getProperty("visitor")) : new SimpleName("
                      TYPE_UNKNOWN")), null),
                                               new SimpleName("
                                                   visitor"),
1995                                           null)),
                  null
                  )
```

```
            );
        }
        else {
          _v0 = new RSLListDefault < MethodDeclaration > (
              new MethodDeclaration(
              new RSLListDefault < Modifier > (Modifier.PUBLIC),
              new SimpleName("accept"),
              PrimitiveType.JAVA_VOID,
              new RSLListDefault < SingleVariableDeclaration >
              (new SingleVariableDeclaration(new RSLListDefault <
                  Modifier > (),
                                             new ReferenceType( (
                                                 translator.
              getProperty("visitor") != null ?
                                             new SimpleName(
                                                 translator.
              getProperty("visitor")) : new SimpleName("
                  TYPE_UNKNOWN")), null),
                                             new SimpleName("
                                                 visitor"),
                                             null)),
              new Block(new RSLListDefault < Statement >
                        (new ExpressionStatement(new
                                                  MethodInvocation(
                                                     new
              SimpleName("visitor"), makeVisitName(id),
                                                  new
                                                      RSLListDefault
                                                       < Expression
                                                       >
              (new ThisExpression(null))))))
              )
              );
        }
      }
      else {
        _v0 = new RSLListDefault < MethodDeclaration > ();
      }
      return _v0;
    }

    public static RSLList < MethodDeclaration >
        makeMainMethodDeclaration(RSLList < Decl > dl) {
      RSLList < MethodDeclaration > _v0 = null;
      if (makeMainMethodDeclaration1(dl).equals(new RSLListDefault
          < Statement > ())) {
        RSLList < MethodDeclaration > _v1 = null;
        _v1 = new RSLListDefault < MethodDeclaration > ();
        _v0 = _v1;
```

```
    }
    else {
      RSLList < MethodDeclaration > _v1 = null;
      _v1 = new RSLListDefault < MethodDeclaration > (
          new MethodDeclaration(
          new RSLListDefault < Modifier >
          (new Modifier[] {Modifier.PUBLIC, Modifier.STATIC}),
          new SimpleName("main"),
          PrimitiveType.JAVA_VOID,
          new RSLListDefault < SingleVariableDeclaration >
          (new SingleVariableDeclaration(new RSLListDefault <
              Modifier > (),
                                         new ArrayType(new
                                             ReferenceType(new
          SimpleName("String"), null)),
                                         new SimpleName("args"),
                                         null)),
          new Block(makeMainMethodDeclaration1(dl))
          )
          );
      _v0 = _v1;
    }
    return _v0;
  }

  public static RSLList < Statement >
      makeMainMethodDeclaration1(RSLList < Decl > dl) {
    RSLList < Statement > _v0 = null;
    if (dl.equals(new RSLListDefault < Decl > ())) {
      RSLList < Statement > _v1 = null;
      _v1 = new RSLListDefault < Statement > ();
      _v0 = _v1;
    }
    else if (dl.hd() instanceof TestDecl) {
      TestDecl td = (TestDecl) dl.hd();
      RSLList < Statement >
          _v1 = makeMainMethodDeclaration2(td.getTestDefList()).
          concat(makeMainMethodDeclaration1(dl.tl()));
      _v0 = _v1;
    }
    else {
      RSLList < Statement > _v1 = makeMainMethodDeclaration1(dl.
          tl());
      _v0 = _v1;
    }
    return _v0;
  }

  public static RSLList < Statement >
```

```
      makeMainMethodDeclaration2(RSLList < TestDef > tdl) {
    RSLList < Statement > _v0 = null;
    if (tdl.equals(new RSLListDefault < TestDef > ())) {
      RSLList < Statement > _v1 = null;
      _v1 = new RSLListDefault < Statement > ();
      _v0 = _v1;
    }
    else {
      RSLList < Statement > _v1 = null;
      _v1 = makeTestDeclaration(tdl.hd()).concat(
          makeMainMethodDeclaration2(tdl.
           tl()));
      _v0 = _v1;
    }
    return _v0;
  }

  public static RSLList < Statement > makeTestDeclaration(
     TestDef td) {
    RSLList < Statement > _v0 = null;
    StringVisitor sv = new StringVisitor();
    td.getValueExpr().accept(sv);
    _v0 = new RSLListDefault < Statement >
        (new ExpressionStatement(new
                                 MethodInvocation(new
                                    QualifiedName(new
        SimpleName("System"), new SimpleName("out")), new
            SimpleName("println"),
                                                    new RSLListDefault
                                                        <
                                                    Expression >
                                                    (new
        InfixExpression(new
                        StringLiteral("[" + td.getId().getText()
                            + "] " +
                                sv.result().replace("\"",
                                   "\\\"") + ": "),
                        InfixOperator.JAVA_INFIX_OP_PLUS,
                        new
                        ParentherizedExpression(makeExpression(
                            td.getValueExpr()))))))));
    return _v0;
  }

  public static JavaType matchType(TypeEvaluator te, boolean
     useWrapper,
                                   boolean useInterface) {
    JavaType _v1 = null;
    if (te instanceof TypeLiteral) {
```

```
      if (te == TypeLiteral.RSLBOOL) {
        if (useWrapper) {
2125      _v1 = new ReferenceType(new SimpleName("Boolean"),
              null);
        }
        else {
          _v1 = PrimitiveType.JAVA_BOOLEAN;
        }
2130    }
      else if (te == TypeLiteral.RSLINT) {
        if (useWrapper) {
          _v1 = new ReferenceType(new SimpleName("Integer"),
              null);
        }
2135    else {
          _v1 = PrimitiveType.JAVA_INT;
        }
      }
      else if (te == TypeLiteral.RSLREAL) {
2140    if (useWrapper) {
          _v1 = new ReferenceType(new SimpleName("Double"), null
              );
        }
        else {
          _v1 = PrimitiveType.JAVA_DOUBLE;
2145    }
      }
      else if (te == TypeLiteral.RSLCHAR) {
        if (useWrapper) {
          _v1 = new ReferenceType(new SimpleName("Character"),
              null);
2150    }
        else {
          _v1 = PrimitiveType.JAVA_CHAR;
        }
      }
2155  else if (te == TypeLiteral.RSLTEXT) {
        _v1 = new ReferenceType(new SimpleName("String"), null);
      }

    }
2160  else if (te instanceof ListTypeExpr) {
      if (te instanceof FiniteListTypeExpr) {
        JavaType _v2 = null;

        if (useInterface) {
2165      _v2 = new ReferenceType(new SimpleName("RSLList"),
                                  (ReferenceType)
                                  matchType( ( (
```

```
                                                FiniteListTypeExpr) te).
                                                    getTypeExpr(), true
                                                        , useInterface));
          }
          else {
            _v2 = new ReferenceType(new SimpleName("RSLListDefault
                "),
                                    (
                                    ReferenceType) matchType( ( (
                                        FiniteListTypeExpr)
                te).getTypeExpr(), true, useInterface));
          }
          _v1 = _v2;
        }
      }
      else if (te instanceof TypeName) {
        TypeName tn = (TypeName) te;
        JavaType _v2 = null;
        _v2 = new ReferenceType(rslId2JavaName(tn.getId()), null);
        _v1 = _v2;
      }
      else if (te instanceof Constructor) {
        Constructor c = (Constructor) te;
        JavaType _v2 = null;
        _v2 = new ReferenceType(rslId2JavaName(c.getId()), null);
        _v1 = _v2;
      }
      else if (te instanceof RecordPattern) {
        RecordPattern rp = (RecordPattern) te;
        JavaType _v2 = null;
        _v2 = new ReferenceType(rslId2JavaName(rp), null);
        _v1 = _v2;
      }
      else if (te instanceof FunctionResultDescription) {
        FunctionResultDescription frd = (FunctionResultDescription
            ) te;
        JavaType _v2 = null;
        _v2 = matchType(frd.getTypeExpr(), useWrapper,
            useInterface);
        _v1 = _v2;
      }
      else {
        _v1 = new ReferenceType(new SimpleName("TYPE_UNKNOWN: " +
            te), null);
      }
      return _v1;
    }

    public static SimpleName rslId2JavaName(Id id) {
```

```
2210      return new SimpleName(id.getText());
        }

        public static SimpleName rslId2JavaName(Pattern pattern) {
          SimpleName _v1 = null;
2215      if (pattern instanceof NamePattern) {
            _v1 = rslId2JavaName( (NamePattern) pattern);
          }
          else if (pattern instanceof RecordPattern) {
            _v1 = rslId2JavaName( (RecordPattern) pattern);
2220      }
          return _v1;
        }

        public static SimpleName rslId2JavaName(NamePattern
            namePattern) {
2225      return new SimpleName(namePattern.getId().getText());
        }

        public static SimpleName rslId2JavaName(RecordPattern
            recordPattern) {
          return new SimpleName(recordPattern.getValueOrVariableName()
              .getId().
2230                                getText());
        }

        public static SimpleName makeVisitName(Id id) {
          return new SimpleName("visit" + id.getText());
2235    }

        /*
          public static void javaast2java(JavaAst javaast,
              BufferedWriter bw) {
          if(javaast != null) {
2240      StringJavaVisitor visitor = new StringJavaVisitor();
          javaast.accept(visitor);
          System.out.println("StringJavaVisitor over Java:\n" +
              visitor.result());
          try {
          bw.write(visitor.result(), 0, visitor.result().length());
2245      bw.flush();
          bw.close();
          }
          catch(IOException ioe) {
          System.out.println("Error writing to file: " + ioe);
2250      System.exit(1);
          }
          System.out.println("Java written to file");
          }
```

```
    else {
2255  System.out.println("No abstract java generated!!!");
    }
    }
   */

2260 public void javaast2java(JavaAst javaast, boolean
       generateFiles,
                            boolean printJava) {
   if (javaast != null) {
     String extraMethods = null;
     StringBuffer fileContent = new StringBuffer();
2265   if (this.getProperty("extraMethodFile") != null) {
       try {
         BufferedReader br = new BufferedReader(new FileReader(
             this.
              getProperty("extraMethodFile")));
         String temp = br.readLine();
2270     while (temp != null) {
           fileContent.append(temp + "\n");
           temp = br.readLine();
         }
       }
2275   catch (FileNotFoundException e) {
         System.out.println("File " + this.getProperty("
             extraMethodFile") +
                            " not found!");
         System.exit(1);
       }
2280   catch (IOException ioe) {
         System.out.println("IOException: " + ioe.getMessage())
             ;
         System.exit(1);
       }
       extraMethods = fileContent.toString();
2285   }
       StringJavaVisitor visitor = new StringJavaVisitor(
         generateFiles,
          this.getProperty("writeFilesDir"), extraMethods);

       //System.out.println("writeFiles: " +
2290   //(new Boolean(this.getProperty("writeFiles"))).
         booleanValue());
       //System.out.println("writeFilesDir: " + this.getProperty
         ("writeFilesDir"));

       javaast.accept(visitor);
       if (printJava) {
2295     System.out.println("StringJavaVisitor over Java:\n" +
```

```
            visitor.result());
        }
      }
      else {
        System.out.println("No java generated!!!");
      }
    }

    public void translate(String filename, boolean generateFiles,
                          boolean printJava) {
      FileReader fr = null;
      try {
        fr = new FileReader(filename + ".rsl");
      }
      catch (FileNotFoundException e) {
        System.out.println("File " + filename + ".rsl not found!")
            ;
        System.exit(1);
      }
      try {
        RSLAst rslast = rsl2rslast(fr);
        /*
          System.out.println("-------------------------------------");
          System.out.println("ARSL:\n" + rslast);
          System.out.println("-------------------------------------");
         */

        ParentVisitor pVisitor = new ParentVisitor();
        /*
          System.out.println("-------------------------------------");
          System.out.println("Setting parent values of AST");
          System.out.println("-------------------------------------");
         */
        rslast.accept(pVisitor);

        if ( (new Boolean(translator.getProperty("printRSL"))).
            booleanValue()) {
          StringVisitor sVisitor = new StringVisitor();
          rslast.accept(sVisitor);
          String result = sVisitor.result();
          System.out.println("-------------------------------------");
          System.out.println("StringVisitor over ARSL:\n" + result
              );
          System.out.println("-------------------------------------");
        }
        /*
          TypeDecorateVisitor tdVisitor =
         new TypeDecorateVisitor(map, recordMapType, recordMapId,
             functionMapType);
```

```
2340          rslast.accept(tdVisitor);
              System.out.println("----------BINDINGS----------");
              System.out.println(tdVisitor.getMap());
              System.out.println("----------------------------------");

2345          TypeDecorateVisitor tdVisitor2 =
             new TypeDecorateVisitor(map, recordMapType, recordMapId,
                functionMapType);
              rslast.accept(tdVisitor2);
              System.out.println("----------BINDINGS----------");
              System.out.println(tdVisitor2.getMap());
2350          System.out.println("----------------------------------");

              System.out.println("----------BINDINGS----------");
              Iterator i = tdVisitor2.getMap().entrySet().iterator();
              while(i.hasNext()) {
2355          System.out.println(i.next());
              System.out.println("----------------------------------");
              }
              System.out.println(tdVisitor2.getMap());
              System.out.println("----------------------------------");
2360         */

            /*JavaAst javaAst = TRANSLATOR_MODULE.rslast2javaast(
                rslast);*/
            JavaAst javaAst = rslast2javaast(rslast);

2365        /*
              System.out.println("----------------------------------");
              System.out.println("JAVAAST:\n" + javaAst);
              System.out.println("----------------------------------");
             */
2370        /*
              BufferedWriter bw = null;
              try {
              bw = new BufferedWriter(new FileWriter(filename + ".java
                  "));
              }
2375          catch(IOException ioe) {
              System.out.println("Errors opening file: " + ioe);
              }

              javaast2java(javaAst, bw);
              System.out.println("----------------------------------");
2380         */
            javaast2java(javaAst, generateFiles, printJava);

            //System.out.println("----------BINDINGS----------");
            //System.out.println(map);
2385
```

```
      //System.out.println("-----------------------------------");
    }
    catch (Exception e) {
      e.printStackTrace();
      System.out.println("Exiting with status 1");
      System.exit(1);
    }
  }

  public String getProperty(String key) {
    return this.properties.getProperty(key);
  }

  public static void main(String[] args) {
    Translator t0 = null;
    if (args.length == 1) {
      t0 = new Translator();
    }
    else if (args.length == 2) {
      t0 = new Translator(args[1]);
    }
    else {
      System.out.println(
          "USAGE: translator.Translator
              RSL_FILE_WITHOUT_EXTENSION [properties-file]");
      System.exit(1);
    }
    translator = t0;
    translator.translate(args[0],
                         (new Boolean(translator.getProperty("
                             writeFiles"))).
                         booleanValue(),
                         (new Boolean(translator.getProperty("
                             printJava"))).
                         booleanValue());
  }
}
```

E.2 translator.rsllib

E.2.1 RSLList.java

```
package translator.rsllib;

import java.util.*;

public interface RSLList<E> {
```

```
    public RSLList<E> listComp(RSLExpression<E> e,
                               Testable<E> t);

    public RSLList<E> concat(RSLList<E> list2);

    public E hd();

    public RSLList<E> tl();

    public E get(int index);

    public RSLList<E> set(int index, E e);

    public int len();

    public RSLSet<E> elems();

    public RSLSet<Integer> inds();

    List<E> getList();
}
```

E.2.2 RSLListDefault.java

```
package translator.rsllib;

import java.util.*;

public class RSLListDefault<E> implements RSLList<E> {

    public ArrayList<E> list;

    public RSLListDefault() {
        this.list = new ArrayList<E>();
    }

    public RSLListDefault(E e) {
        this.list = new ArrayList<E>();
        this.list.add(e);
    }

    public RSLListDefault(E[] init) {
        this.list = new ArrayList<E>(Arrays.asList(init));
    }

    public RSLListDefault(RSLList<E> init) {
        this.list = new ArrayList<E>();
        for(E e : init.getList())
            this.list.add(e);
```

```
    }

    public RSLListDefault(Integer low, Integer high) {
        this.list = new ArrayList<E>();

        for(int i = low, j = 0; i <= high; i++, j++) {
            this.list.add((E)( new Integer(i)));
        }
    }

    public RSLListDefault<E> listComp(RSLExpression<E>
                                      expression,
                                      Testable<E>
                                      testable) {
        ArrayList<E> elements = new ArrayList<E>();
        for(E e : this.getList()) {
            if(testable.test(e))
                elements.add(expression.action(e));
        }
        RSLListDefault<E> result = new RSLListDefault<E>();
        result.list = elements;
        return result;
    }

    public RSLList<E> concat(RSLList<E> list2) {
        /*list.addAll(list2.getList());
          return this;
        */
        ArrayList<E> newList = new ArrayList<E>(list);
        newList.addAll(list2.getList());
        RSLListDefault<E> result = new RSLListDefault<E>();
        result.list = newList;
        return result;
    }

    public E hd() {
        return list.get(0);
    }

    public RSLListDefault<E> tl() {
        ArrayList<E> list2 = new ArrayList<E>(list);
        RSLListDefault<E> result = new RSLListDefault<E>();
        result.list = list2;
        result.list.remove(0);
        return result;
    }

    public E get(int index) {
        return list.get(index-1);
```

```
75      }

        public RSLList<E> set(int index, E e) {
            list.set(index-1, e);
            return this;
80      }

        public int len() {
            return list.size();
        }
85
        public RSLSet<E> elems() {
            return new RSLSetDefault(list.toArray());
        }

90      public RSLSet<Integer> inds() {
            return new RSLSetDefault(1, len());
        }

        public List<E> getList() {
95          return list;
        }

        public boolean equals(Object o) {
            if( o instanceof RSLListDefault) {
100             return list.equals(((RSLListDefault) o).
                                            getList());
            }
            else {
                return false;
105         }
        }

        public String toString() {
            StringBuffer result = new StringBuffer();
110         result.append("<.");
            for(E e : list) {
                result.append(e.toString());
                result.append(",");
            }
115         if(!list.isEmpty())
                result.delete(result.length() - 1,
                              result.length());
            result.append(".>");
            return result.toString();
120     }
    }
```

E.2.3 RSLSet.java

```
package translator.rsllib;

import java.util.*;

public interface RSLSet<E> {
    public Set<E> getSet();

    public boolean isIn(E element);

    public boolean isNotIn(E element);

    public RSLSet<E> union(RSLSet<E> set2);

    public RSLSet<E> intersect(RSLSet<E> set2);

    public RSLSet<E> difference(RSLSet<E> set2);

    public boolean superSet(RSLSet<E> set2);

    public boolean properSuperSet(RSLSet<E> set2);

    public boolean subSet(RSLSet<E> set2);

    public boolean properSubSet(RSLSet<E> set2);

    public int card();
}
```

E.2.4 RSLSetDefault.java

```
package translator.rsllib;

import java.util.*;

public class RSLSetDefault<E> implements RSLSet<E> {
    private HashSet<E> set;

    public RSLSetDefault() {
        set = new HashSet<E>();
    }

    public RSLSetDefault(E e) {
        this.set = new HashSet<E>();
        this.set.add(e);
    }

    public RSLSetDefault(E[] init) {
```

```
        this.set = new HashSet<E>(Arrays.asList(init));
    }

    public RSLSetDefault(Integer low, Integer high) {
        this.set = new HashSet<E>();

        for(int i = low, j = 0; i <= high; i++, j++) {
            this.set.add((E)( new Integer(i)));
        }
    }


    RSLSetDefault(Collection<E> collection) {
        set = new HashSet<E>(collection);
    }

    public Set<E> getSet() {
        return set;
    }

    public boolean isIn(E element) {
        return set.contains(element);
    }

    public boolean isNotIn(E element) {
        return !set.contains(element);
    }

    public RSLSet<E> union(RSLSet<E> set2) {
        HashSet<E> newSet = new HashSet<E>(set);
        newSet.addAll(set2.getSet());
        RSLSetDefault<E> result = new RSLSetDefault<E>();
        result.set = newSet;
        return result;
    }

    public RSLSet<E> intersect(RSLSet<E> set2) {
        HashSet<E> newSet = new HashSet<E>(set);
        newSet.retainAll(set2.getSet());
        RSLSetDefault<E> result = new RSLSetDefault<E>();
        result.set = newSet;
        return result;
    }

    public RSLSet<E> difference(RSLSet<E> set2) {
        HashSet<E> newSet = new HashSet<E>(set);
        newSet.removeAll(set2.getSet());
        RSLSetDefault<E> result = new RSLSetDefault<E>();
        result.set = newSet;
```

```
        return result;
    }

    public boolean superSet(RSLSet<E> set2) {
        return this.set.containsAll(set2.getSet());
    }

    public boolean properSuperSet(RSLSet<E> set2) {
        return this.superSet(set2) &&
               this.card() < set2.card();
    }


    public boolean subSet(RSLSet<E> set2) {
        return set2.getSet().containsAll(this.set);
    }

    public boolean properSubSet(RSLSet<E> set2) {
        return this.subSet(set2) &&
               this.card() > set2.card();
    }

    public int card() {
        return set.size();
    }

    public String toString() {
        StringBuffer result = new StringBuffer();
        result.append("{");
        for(E e : set) {
            result.append(e.toString());
            result.append(",");
        }
        if(!set.isEmpty())
            result.delete(result.length() - 1,
                          result.length());
        result.append("}");
        return result.toString();

    }
}
```

E.2.5 RSLMap.java

```
package translator.rsllib;

import java.util.*;

public interface RSLMap<K, V> {
```

```
    public RSLSet<K> dom();

    public RSLSet<V> rng();

    public RSLMap<K,V> override(RSLMap<K,V> newValues);

    public V get(K key);

    public RSLMap<K,V> union(RSLMap<K,V> addedValues);

    public RSLMap<K,V> restrictBy(RSLSet<K> keys);

    public RSLMap<K,V> restrictTo(RSLSet<K> keys);

    //public RSLMap<?,V> compose(RSLMap<K,?> innerMap);

     public RSLMap compose(RSLMap innerMap);

    public Map<K,V> getMap();
}
```

E.2.6 RSLMapDefault.java

```
package translator.rsllib;

import java.util.*;

public class RSLMapDefault<K,V> implements RSLMap<K,V> {

    private HashMap<K,V> map;

    public RSLMapDefault() {
        this.map = new HashMap<K,V>();
    }

    public RSLMapDefault(K key, V value) {
        this.map = new HashMap<K,V>();
        this.map.put(key, value);
    }

    public RSLMapDefault(K[] keys, V[] values) {
        if(keys.length != values.length)
            throw new IllegalArgumentException(
            "Key and value arrays are " +
            "of different lengths.");
        this.map = new HashMap<K,V>();
        for(int i = 0; i < keys.length; i++) {
            this.map.put(keys[i], values[i]);
```

```
        }
    }

    public RSLSet<K> dom() {
        return new RSLSetDefault<K>(map.keySet());
    }

    public RSLSet<V> rng() {
        return new RSLSetDefault<V>(map.values());
    }

    public RSLMap<K,V> override(RSLMap<K,V> newValues) {
        RSLMapDefault<K,V> result = new RSLMapDefault<K,V>();
        result.map = new HashMap<K,V>(this.map);
        result.map.putAll(newValues.getMap());
        return result;
    }

    public V get(K key) {
        return map.get(key);
    }

    public RSLMap<K,V> union(RSLMap<K,V> addedValues) {
        this.map.putAll(addedValues.getMap());
        return this;
    }

    public RSLMap<K,V> restrictBy(RSLSet<K> keys) {
        RSLMapDefault<K,V> result =
                new RSLMapDefault<K,V>();
        result.map = new HashMap<K,V>(this.map);

        result.map.keySet().removeAll(keys.getSet());
        return result;
    }

    public RSLMap<K,V> restrictTo(RSLSet<K> keys) {
        RSLMapDefault<K,V> result =
                new RSLMapDefault<K,V>();
        result.map = new HashMap<K,V>(this.map);

        result.map.keySet().retainAll(keys.getSet());
        return result;
    }


    public RSLMap compose(RSLMap innerMap) {
        RSLMap result = new RSLMapDefault();
```

```
75          Iterator iterator =
                    innerMap.getMap().entrySet().iterator();
            while(iterator.hasNext()) {
                Map.Entry elem = (Map.Entry) iterator.next();

80              if(this.getMap().containsKey((K)elem.
                                              getValue())) {
                    result.getMap().put(elem.getKey(),
                    (V)this.getMap().get(
                            innerMap.get(elem.getKey())));
85              }

            }
            return result;
        }
90
        public Map<K,V> getMap() {
            return map;
        }

95      public String toString() {
            return map.toString();
        }
    }
```

E.3 translator.rslast

E.3.1 ApplicationExpr.java

```
    package translator.rslast;

    import translator.lib.*;
    import translator.rsllib.*;
5
    public class ApplicationExpr
        extends ValueExpr
        implements Element {
      private ValueExpr valueExpr;
10    private RSLList < ValueExpr > optionalValueExprList;

      public ApplicationExpr(ValueExpr valueExpr,
                             RSLListDefault < ValueExpr >
                                optionalValueExprList) {
        this.valueExpr = valueExpr;
15      this.optionalValueExprList = optionalValueExprList;
      }

      public ValueExpr getValueExpr() {
```

```
    return valueExpr;
  }

  public void setValueExpr(ValueExpr valueExpr) {
    this.valueExpr = valueExpr;
  }

  public RSLList < ValueExpr > getOptionalValueExprList() {
    return optionalValueExprList;
  }

  public void setOptionalValueExprList(RSLList < ValueExpr >
                                       optionalValueExprList) {
    this.optionalValueExprList = optionalValueExprList;
  }

  public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("ApplicationExpr(");
    result.append(valueExpr.toString());
    if (!optionalValueExprList.getList().isEmpty()) {
      result.append(", ValueExprList(");
      for (ValueExpr ve : optionalValueExprList.getList()) {
        result.append(ve.toString());
        result.append(", ");
      }
      result.delete(result.length() - 2, result.length());
      result.append(")");
    }
    result.append(")");
    return result.toString();
  }

  public void accept(Visitor visitor) {
    visitor.visitApplicationExpr(this);
  }
}
```

E.3.2 BasicClassExpr.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;

public class BasicClassExpr extends ClassExpr implements Element
    {
    private RSLList<Decl> declList;
```

```
    public BasicClassExpr(RSLList<Decl> declList) {
        this.declList = declList;
    }

    public RSLList<Decl> getDeclList() {
        return declList;
    }

    public void setDeclList(RSLList<Decl> declList) {
        this.declList = declList;
    }

    public String toString() {
        StringBuffer result = new StringBuffer("BasicClassExpr(\
            n");
        for(Decl decl : declList.getList()) {
            result.append(decl.toString());
            result.append(", \n");
        }
        if(declList.getList().isEmpty())
            result.delete(result.length() - 3, result.length());
        result.append(")");
        return result.toString();
    }

    public void accept(Visitor visitor) {
        visitor.visitBasicClassExpr(this);
    }
}
```

E.3.3 Binding.java

```
package translator.rslast;

import translator.lib.*;

public abstract class Binding extends RSLElement {

    public abstract void accept(Visitor visitor);

    public abstract String toString();
}
```

E.3.4 CaseBranch.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;
```

```
5
public class CaseBranch extends RSLElement {
    private Pattern pattern;
    private ValueExpr valueExpr;

10  public CaseBranch(Pattern pattern, ValueExpr valueExpr) {
        this.pattern = pattern;
        this.valueExpr = valueExpr;
    }

15  public Pattern getPattern() {
        return pattern;
    }

    public void setPattern(Pattern pattern) {
20      this.pattern = pattern;
    }

    public ValueExpr getValueExpr() {
        return valueExpr;
25  }

    public void setValueExpr(ValueExpr valueExpr) {
        this.valueExpr = valueExpr;
    }
30
    public String toString() {
        StringBuffer result = new StringBuffer();
        result.append("CaseBranch(");
        result.append(pattern.toString());
35      result.append(", ");
        result.append(valueExpr.toString());
        result.append(")");
        return result.toString();
    }
40
    public void accept(Visitor visitor) {
        visitor.visitCaseBranch(this);
    }
}
```

E.3.5 CaseExpr.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;
5
public class CaseExpr extends StructuredExpr implements Element
```

```
    {
    private ValueExpr condition;
    private RSLList<CaseBranch> caseBranchList;

    public CaseExpr(ValueExpr condition, RSLList<CaseBranch>
        caseBranchList) {
        this.condition = condition;
        this.caseBranchList = caseBranchList;
    }

    public ValueExpr getCondition() {
        return condition;
    }

    public void setCondition(ValueExpr condition) {
        this.condition = condition;
    }

    public RSLList<CaseBranch> getCaseBranchList() {
        return caseBranchList;
    }

    public void setCaseBranchList(RSLList<CaseBranch>
        caseBranchList) {
        this.caseBranchList = caseBranchList;
    }

    public String toString() {
        StringBuffer result = new StringBuffer();
        result.append("CaseExpr(");
        result.append(condition.toString());
        result.append(", CaseBranchList(");
        for(CaseBranch cb : caseBranchList.getList()) {
            result.append(cb.toString());
        }
        result.append(")");
        result.append(")");
        return result.toString();
    }

    public void accept(Visitor visitor) {
        visitor.visitCaseExpr(this);
    }
}
```

E.3.6 ClassExpr.java

```
package translator.rslast;
```

```
import translator.lib.*;

public abstract class ClassExpr extends RSLElement {

    public abstract void accept(Visitor visitor);

    public abstract String toString();
}
```

E.3.7 ComponentKind.java

```
package translator.rslast;

import translator.lib.*;

public class ComponentKind
    extends RSLElement {
  private Destructor optionalDestructor;
  private TypeExpr typeExpr;
  private Id optionalReconstructor;

  public ComponentKind(Destructor optionalDestructor, TypeExpr
      typeExpr,
                       Id optionalReconstructor) {
    this.optionalDestructor = optionalDestructor;
    this.typeExpr = typeExpr;
    this.optionalReconstructor = optionalReconstructor;
  }

  public Destructor getOptionalDestructor() {
    return optionalDestructor;
  }

  public void setOptionalDestructor(Destructor
      optionalDestructor) {
    this.optionalDestructor = optionalDestructor;
  }

  public TypeExpr getTypeExpr() {
    return typeExpr;
  }

  public void setTypeExpr(TypeExpr typeExpr) {
    this.typeExpr = typeExpr;
  }

  public Id getOptionalReconstructor() {
    return optionalReconstructor;
  }
```

```
  public void setOptionalReconstructor(Id optionalReconstructor)
      {
    this.optionalReconstructor = optionalReconstructor;
  }

  public String toString() {
    StringBuffer result = new StringBuffer("ComponenentKind(");
    if (optionalDestructor != null) {
      result.append(optionalDestructor);
    }
    else {
      result.append("NoOptionalDestructor()");
    }
    result.append(", ");
    result.append(typeExpr.toString());
    if (optionalReconstructor != null) {
      result.append(optionalReconstructor.toString());
    }
    else {
      result.append("NoOptionalReconstructor()");
    }
    result.append(")");
    return result.toString();
  }

  public void accept(Visitor visitor) {
    visitor.visitComponentKind(this);
  }
}
```

E.3.8 Constructor.java

```
package translator.rslast;

import translator.lib.*;

public class Constructor extends Variant implements
    TypeEvaluator {
    private Id id;

    public Constructor(Id id) {
        this.id = id;
    }

    public Id getId() {
        return id;
    }

```

```
    public void setId(Id id) {
        this.id = id;
    }

    public String toString() {
        return "Constructor(" + id.toString() + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitConstructor(this);
    }
}
```

E.3.9 Decl.java

```
package translator.rslast;

import translator.lib.*;

public abstract class Decl extends RSLElement {

    public abstract void accept(Visitor visitor);

    public abstract String toString();
}
```

E.3.10 Destructor.java

```
package translator.rslast;

import translator.lib.*;

public class Destructor extends RSLElement implements
    TypeEvaluator {
    private Id id;

    public Destructor(Id id) {
        this.id = id;
    }

    public Id getId() {
        return id;
    }

    public void setId(Id id) {
        this.id = id;
    }

    public String toString() {
```

```
        return "Destructor(" + id.toString() + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitDestructor(this);
    }
}
```

E.3.11 DisambiguationExpr.java

```
package translator.rslast;

import translator.lib.*;

public class DisambiguationExpr extends ValueExpr implements
    Element {
    private ValueExpr valueExpr;
    private TypeExpr typeExpr;

    public DisambiguationExpr(ValueExpr valueExpr, TypeExpr
        typeExpr) {
        this.valueExpr = valueExpr;
        this.typeExpr = typeExpr;
    }

    public ValueExpr getValueExpr() {
        return valueExpr;
    }

    public void setValueExpr(ValueExpr valueExpr) {
        this.valueExpr = valueExpr;
    }

    public TypeExpr getTypeExpr() {
        return typeExpr;
    }

    public void setTypeExpr(TypeExpr typeExpr) {
        this.typeExpr = typeExpr;
    }

    public String toString() {
        StringBuffer result = new StringBuffer("
            DisambiguationExpr(");
        result.append(valueExpr.toString());
        result.append(", ");
        result.append(typeExpr.toString());
        result.append(")");
        return result.toString();
```

```
    }


    public void accept(Visitor visitor) {
        visitor.visitDisambiguationExpr(this);
    }
}
```

E.3.12 ElseBranch.java

```
package translator.rslast;

import translator.lib.*;

public class ElseBranch extends RSLElement {
    private ValueExpr valueExpr;

    public ElseBranch(ValueExpr valueExpr) {
        this.valueExpr = valueExpr;
    }

    public ValueExpr getValueExpr() {
        return valueExpr;
    }

    public void setValueExpr(ValueExpr valueExpr) {
        this.valueExpr = valueExpr;
    }

    public String toString() {
        return "elseBranch( " + valueExpr.toString() + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitElseBranch(this);
    }
}
```

E.3.13 ElsifBranch.java

```
package translator.rslast;

import translator.lib.*;

public class ElsifBranch extends RSLElement {
    private ValueExpr condition;
    private ValueExpr valueExpr;
```

```
    public ElsifBranch(ValueExpr condition, ValueExpr valueExpr)
        {
        this.condition = condition;
        this.valueExpr = valueExpr;
    }

    public ValueExpr getCondition() {
        return condition;
    }

    public void setCondition(ValueExpr condition) {
        this.condition = condition;
    }

    public ValueExpr getValueExpr() {
        return valueExpr;
    }

    public void setValueExpr(ValueExpr valueExpr) {
        this.valueExpr = valueExpr;
    }

    public String toString() {
        StringBuffer result = new StringBuffer("ElsifBranch(");
        result.append(condition.toString());
        result.append(", ");
        result.append(valueExpr.toString());
        result.append(")");
        return result.toString();
    }

    public void accept(Visitor visitor) {
        visitor.visitElsifBranch(this);
    }
}
```

E.3.14 EnumeratedListExpr.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;

public class EnumeratedListExpr extends ListExpr {
    private RSLListDefault<ValueExpr> valueExprList;

    public EnumeratedListExpr(RSLListDefault<ValueExpr>
        valueExprList) {
        this.valueExprList = valueExprList;
```

```
    }

    public RSLList<ValueExpr> getValueExprList() {
        return valueExprList;
    }

    public void setValueExprList(RSLListDefault<ValueExpr>
        valueExprList) {
        this.valueExprList = valueExprList;
    }

    public String toString() {
        StringBuffer result = new StringBuffer();
        boolean hasElements = false;
        result.append("EnumeratedListExpr(");
        for(ValueExpr ve : valueExprList.getList()) {
            result.append(ve.toString() + ", ");
            hasElements = true;
        }
        if(hasElements)
            result.delete(result.length() - 2, result.length());
        result.append(")");

        return result.toString();
    }

    public void accept(Visitor visitor) {
        visitor.visitEnumeratedListExpr(this);
    }

}
```

E.3.15 ExplicitFunctionDef.java

```
package translator.rslast;

import translator.lib.*;

public class ExplicitFunctionDef
    extends ValueDef {
  private SingleTyping singleTyping;
  private FormalFunctionApplication formalFunctionApplication;
  private ValueExpr valueExpr;
  private OptionalPrecondition optionalPrecondition;

  public ExplicitFunctionDef(SingleTyping singleTyping,
                             FormalFunctionApplication
                             formalFunctionApplication,
                                ValueExpr valueExpr,
```

```
15                                  OptionalPrecondition
                                       optionalPrecondition) {
    this.singleTyping = singleTyping;
    this.formalFunctionApplication = formalFunctionApplication;
    this.valueExpr = valueExpr;
    this.optionalPrecondition = optionalPrecondition;
20  }

  public SingleTyping getSingleTyping() {
    return singleTyping;
  }
25
  public void setSingleTyping(SingleTyping singleTyping) {
    this.singleTyping = singleTyping;
  }

30  public FormalFunctionApplication getFormalFunctionApplication
      () {
    return formalFunctionApplication;
  }

  public void setFormalFunctionApplication(
      FormalFunctionApplication
35                                          formalFunctionApplication
                                               ) {
    this.formalFunctionApplication = formalFunctionApplication;
  }

  public ValueExpr getValueExpr() {
40    return valueExpr;
  }

  public void setValueExpr(ValueExpr valueExpr) {
    this.valueExpr = valueExpr;
45  }

  public OptionalPrecondition getOptionalPrecondition() {
    return optionalPrecondition;
  }
50
  public void setOptionalPrecondition(OptionalPrecondition
      optionalPrecondition) {
    this.optionalPrecondition = optionalPrecondition;
  }

55  public String toString() {
    StringBuffer result = new StringBuffer("ExplicitFunctionDef
        (\n");
    result.append("\t" + singleTyping.toString() + ",\n");
```

```
      result.append("\t" + formalFunctionApplication.toString() +
          ",\n");
      result.append("\t" + valueExpr.toString() + ",\n");
      result.append("\t" + optionalPrecondition.toString() + "\n)"
          );
      return result.toString();
    }

    public void accept(Visitor visitor) {
      visitor.visitExplicitFunctionDef(this);
    }
  }
```

E.3.16 ExtendingClassExpr.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;

public class ExtendingClassExpr extends ClassExpr implements
    Element {
    private ClassExpr baseClass;
    private ClassExpr extensionClass;

    public ExtendingClassExpr(ClassExpr baseClass, ClassExpr
        extensionClass) {
        this.baseClass = baseClass;
        this.extensionClass = extensionClass;
    }

    public ClassExpr getBaseClass() {
        return baseClass;
    }

    public void setBaseClass(ClassExpr baseClass) {
        this.baseClass = baseClass;
    }

    public ClassExpr getExtensionClass() {
        return extensionClass;
    }

    public void setExtensionClass(ClassExpr extensionClass) {
        this.extensionClass = extensionClass;
    }

    public String toString() {
        StringBuffer result = new StringBuffer("
```

```
            ExtendingClassExpr(\n");
        result.append(baseClass);
        result.append(", ");
        result.append(extensionClass);
        result.append(")");
        return result.toString();
    }

    public void accept(Visitor visitor) {
        visitor.visitExtendingClassExpr(this);
    }
}
```

E.3.17 FiniteListTypeExpr.java

```
package translator.rslast;

import translator.lib.*;

public class FiniteListTypeExpr extends ListTypeExpr implements
    Element {
    private TypeExpr typeExpr;

    public FiniteListTypeExpr(TypeExpr typeExpr) {
        this.typeExpr = typeExpr;
    }

    public TypeExpr getTypeExpr() {
        return typeExpr;
    }

    public void setTypeExpr(TypeExpr typeExpr) {
        this.typeExpr = typeExpr;
    }

    public String toString() {
        return "FiniteListExpr(" + typeExpr + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitFiniteListTypeExpr(this);
    }
}
```

E.3.18 FormalFunctionApplication.java

```
package translator.rslast;

import translator.lib.*;
```

```
5  public abstract class FormalFunctionApplication extends
       RSLElement {

       public abstract String toString();

       public abstract void accept(Visitor visitor);
10 }
```

E.3.19 FormalFunctionParameter.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;
5
public class FormalFunctionParameter extends RSLElement {
    private RSLList<Binding> bindingList;

    public FormalFunctionParameter(RSLList<Binding> bindingList)
        {
10      this.bindingList = bindingList;
    }

    public RSLList<Binding> getBindingList() {
        return bindingList;
15  }

    public void setBindingList(RSLList<Binding> bindingList) {
        this.bindingList = bindingList;
    }
20
    public String toString() {
        StringBuffer result = new StringBuffer();
        boolean hasElements = false;

25      result.append("FormalFunctionParameter(");
        for(Binding binding : bindingList.getList()) {
            result.append(binding.toString() + ", ");
            hasElements = true;
        }
30      if(hasElements) {
            result.delete(result.length() - 2, result.length());
        }
        result.append(")");
        return result.toString();
35  }

    public void accept(Visitor visitor) {
```

```
        visitor.visitFormalFunctionParameter(this);
    }
}
```

E.3.20 FunctionArrow.java

```
package translator.rslast;

import translator.lib.*;

public class FunctionArrow
    extends RSLElement {
  public static final FunctionArrow TOTAL_FUNCTION_ARROW = new
      FunctionArrow(
      "->");
  public static final FunctionArrow PARTIAL_FUNCTION_ARROW = new
      FunctionArrow(
      "-m->");

  private String text;

  private FunctionArrow(String text) {
    this.text = text;
  }

  public String getText() {
    return text;
  }

  public String toString() {
    return "FunctionArrow." +
        (this == TOTAL_FUNCTION_ARROW ? "TOTAL_FUNCTION_ARROW" :
         "PARTIAL_FUNCTION_ARROW");
  }

  public void accept(Visitor visitor) {
    visitor.visitFunctionArrow(this);
  }
}
```

E.3.21 FunctionResultDescription.java

```
package translator.rslast;

import translator.lib.*;

public class FunctionResultDescription
    extends RSLElement
    implements TypeEvaluator {
```

```
  private OptionalAccessDescription optionalAccessDescription;
  private TypeExpr typeExpr;

  public FunctionResultDescription(OptionalAccessDescription
                                   optionalAccessDescription,
                                      TypeExpr typeExpr) {
    this.optionalAccessDescription = optionalAccessDescription;
    this.typeExpr = typeExpr;
  }

  public OptionalAccessDescription getOptionalAccessDescription
     () {
    return optionalAccessDescription;
  }

  public void setOptionalAccessDescription(
     OptionalAccessDescription
                                           optionalAccessDescription
                                              ) {
    this.optionalAccessDescription = optionalAccessDescription;
  }

  public TypeExpr getTypeExpr() {
    return typeExpr;
  }

  public void setTypeExpr(TypeExpr typeExpr) {
    this.typeExpr = typeExpr;
  }

  public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("FunctionResultDescription(");
    if (optionalAccessDescription != null) {
      result.append(optionalAccessDescription.toString() + ", ")
          ;
    }
    result.append(typeExpr.toString());
    result.append(")");
    return result.toString();
  }

  public void accept(Visitor visitor) {
    visitor.visitFunctionResultDescription(this);
  }

}
```

E.3.22 FunctionTypeExpr.java

```
package translator.rslast;

import translator.lib.*;

public class FunctionTypeExpr
    extends TypeExpr
    implements Element {
  private TypeExpr typeExpr;
  private FunctionArrow functionArrow;
  private FunctionResultDescription functionResultDescription;

  public FunctionTypeExpr(TypeExpr typeExpr, FunctionArrow
      functionArrow,
                          FunctionResultDescription
                              functionResultDescription) {
    this.typeExpr = typeExpr;
    this.functionArrow = functionArrow;
    this.functionResultDescription = functionResultDescription;
  }

  public TypeExpr getTypeExpr() {
    return typeExpr;
  }

  public void setTypeExpr(TypeExpr typeExpr) {
    this.typeExpr = typeExpr;
  }

  public FunctionArrow getFunctionArrow() {
    return functionArrow;
  }

  public void setFunctionArrow(FunctionArrow functionArrow) {
    this.functionArrow = functionArrow;
  }

  public FunctionResultDescription getFunctionResultDescription
      () {
    return functionResultDescription;
  }

  public void setFunctionResultDescription(
      FunctionResultDescription
                                           functionResultDescription
                                               ) {
    this.functionResultDescription = functionResultDescription;
  }
```

```
  public String toString() {
45    StringBuffer result = new StringBuffer();
      result.append("FunctionTypeExpr(");
      result.append(typeExpr.toString() + ", ");
      result.append(functionArrow.toString() + ", ");
      result.append(functionResultDescription.toString());
50    result.append(")");
      return result.toString();
    }

    public void accept(Visitor visitor) {
55    visitor.visitFunctionTypeExpr(this);
    }
  }
```

E.3.23 Id.java

```
package translator.rslast;

import translator.lib.*;

5 public class Id extends IdOrOp implements Element, Comparable<Id
    >{
    private String text;

    public Id(String text) {
        this.text = text;
10  }

    public String getText() {
        return text;
    }
15
    public void setText(String text) {
        this.text = text;
    }

20  public String toString() {
        return "Id(" + text + ")";
    }

    public int compareTo(Id id) {
25      return text.compareTo(id.getText());
    }

    public int hashCode() {
        return text.hashCode();
30  }
```

```
    public boolean equals(Object object) {
        if(object instanceof Id)
            return text.equals(((Id) object).getText());
        return false;
    }

    public void accept(Visitor visitor) {
        visitor.visitId(this);
    }
}
```

E.3.24 IdApplication.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;

public class IdApplication
    extends FormalFunctionApplication
    implements Element {
  private Id id;
  private RSLList < FormalFunctionParameter >
    formalFunctionParameters;

  public IdApplication(Id id,
                       RSLList < FormalFunctionParameter >
                       formalFunctionParameters) {
    this.id = id;
    this.formalFunctionParameters = formalFunctionParameters;
  }

  public Id getId() {
    return id;
  }

  public void setId(Id id) {
    this.id = id;
  }

  public RSLList < FormalFunctionParameter >
    getFormalFunctionParameters() {
    return formalFunctionParameters;
  }

  public void setFormalFunctionParameters(RSLList <
    FormalFunctionParameter >
                                          formalFunctionParameters
```

```
                                                    ) {
    this.formalFunctionParameters = formalFunctionParameters;
  }

  public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("IdApplication(" + id + ", ");
    result.append("FormalFunctionParameterList(");
    for (FormalFunctionParameter parameter :
        formalFunctionParameters.getList()) {
      result.append(parameter.toString() + ", ");
    }
    if (!formalFunctionParameters.getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")");
    result.append(")");
    return result.toString();
  }

  public void accept(Visitor visitor) {
    visitor.visitIdApplication(this);
  }
}
```

E.3.25 IdOrOp.java

```
package translator.rslast;

import translator.lib.*;

public abstract class IdOrOp extends Binding {

    public abstract String toString();

    public abstract void accept(Visitor visitor);
}
```

E.3.26 IfExpr.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;

public class IfExpr
    extends StructuredExpr
    implements Element {
  private ValueExpr condition;
```

```
10    private ValueExpr valueExpr;
      private RSLList < ElsifBranch > elsifBranchList;
      private ElseBranch elseBranch;

      public IfExpr(ValueExpr condition, ValueExpr valueExpr,
15                  RSLList < ElsifBranch > elsifBranchList,
                        ElseBranch elseBranch) {
        this.condition = condition;
        this.valueExpr = valueExpr;
        this.elsifBranchList = elsifBranchList;
        this.elseBranch = elseBranch;
20    }

      public ValueExpr getCondition() {
        return condition;
      }
25
      public void setCondition(ValueExpr condition) {
        this.condition = condition;
      }

30    public ValueExpr getValueExpr() {
        return valueExpr;
      }

      public void setValueExpr(ValueExpr valueExpr) {
35      this.valueExpr = valueExpr;
      }

      public RSLList < ElsifBranch > getElsifBranchList() {
        return elsifBranchList;
40    }

      public void setElsifBranchList(RSLList < ElsifBranch >
         elsifBranchList) {
        this.elsifBranchList = elsifBranchList;
      }
45
      public ElseBranch getElseBranch() {
        return elseBranch;
      }

50    public void setElseBranch(ElseBranch elseBranch) {
        this.elseBranch = elseBranch;
      }

      public String toString() {
55      StringBuffer result = new StringBuffer();
        result.append("IfExpr(");
```

```
      result.append(condition.toString());
      result.append(", ");
      result.append(valueExpr.toString());
      result.append(", ElsifBranchList(");
      for (ElsifBranch elsif : elsifBranchList.getList()) {
        result.append(elsif.toString());
      }
      result.append("),");
      if (elseBranch != null) {
        result.append(elseBranch.toString());
      }
      else {
        result.append("noElseBranch()");
      }
      result.append(")");
      return result.toString();
    }

    public void accept(Visitor visitor) {
      visitor.visitIfExpr(this);
    }
  }
```

E.3.27 InfixExpr.java

```
package translator.rslast;

import translator.lib.*;

public abstract class InfixExpr extends ValueExpr implements
    Element {

    public abstract String toString();

    public abstract void accept(Visitor visitor);
}
```

E.3.28 LibModule.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;

public class LibModule extends RSLElement {
    private RSLList<Id> contextList;
    private SchemeDef scheme;
```

```
10      public LibModule(RSLList<Id> contextList, SchemeDef scheme)
            {
            this.contextList = contextList;
            this.scheme = scheme;
        }

15      public RSLList<Id> getContextList() {
            return contextList;
        }

        public void setContextList(RSLList<Id> contextList) {
20          this.contextList = contextList;
        }

        public SchemeDef getSchemeDef() {
            return scheme;
25      }

        public void setSchemeDef(SchemeDef scheme) {
            this.scheme = scheme;
        }
30
        public String toString() {
            StringBuffer result = new StringBuffer();
            result.append("LibModule(\n");
            for(Id id : contextList.getList()) {
35              result.append(id.toString());
                result.append(", ");
            }
            if(!contextList.getList().isEmpty())
                result.delete(result.length() - 2, result.length());
40          result.append(")");
            result.append(scheme.toString());
            result.append(")");
            return result.toString();
        }
45
        public void accept(Visitor visitor) {
            visitor.visitLibModule(this);
        }

50  }
```

E.3.29 ListExpr.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;
```

```
5
public abstract class ListExpr extends ValueExpr  {
    public abstract String toString();
    public abstract void accept(Visitor visitor);
}
```

E.3.30 ListTypeExpr.java

```
package translator.rslast;

import translator.lib.*;

5 public abstract class ListTypeExpr extends TypeExpr implements
    Element {

    public abstract String toString();

    public abstract void accept(Visitor visitor);
10 }
```

E.3.31 NamePattern.java

```
package translator.rslast;

import translator.lib.*;

5 public class NamePattern extends RSLElement implements Pattern,
    Element {
    private Id id;
    private Id valueInitializer;

    public NamePattern(Id id) {
10      this.id = id;
    }

    public Id getId() {
        return id;
15  }

    public void setId(Id id) {
        this.id = id;
    }
20
    public void setValueInitializer(Id valueInitializer) {
        this.valueInitializer = valueInitializer;
    }

25  public Id getValueInitializer() {
        return valueInitializer;
```

```
    }

    public String toString() {
        return "NamePattern(" + id + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitNamePattern(this);
    }
}
```

E.3.32 NoAccessDescription.java

```
package translator.rslast;

import translator.lib.*;

public class NoAccessDescription extends
    OptionalAccessDescription
    implements Element {


    public String toString() {
        return "NoAccessDescription()";
    }

    public void accept(Visitor visitor) {
        visitor.visitNoAccessDescription(this);
    }

}
```

E.3.33 NoPrecondition.java

```
package translator.rslast;

import translator.lib.*;

public class NoPrecondition extends OptionalPrecondition
   implements Element {


    public String toString() {
        return "NoPrecondition()";
    }


    public void accept(Visitor visitor) {
        visitor.visitNoPrecondition(this);
```

```
15      }
}
```

E.3.34 OptionalAccessDescription.java

```
package translator.rslast;

import translator.lib.*;

5 public abstract class OptionalAccessDescription extends
    RSLElement {

    public abstract void accept(Visitor visitor);
}
```

E.3.35 OptionalPrecondition.java

```
package translator.rslast;

import translator.lib.*;

5 public abstract class OptionalPrecondition extends RSLElement {

    public abstract void accept(Visitor visitor);
}
```

E.3.36 Pattern.java

```
package translator.rslast;

import translator.lib.*;

5 public interface Pattern extends Element{

    public RSLElement getParent();

    public void setParent(RSLElement element);
10 }
```

E.3.37 Precondition.java

```
package translator.rslast;

import translator.lib.*;

5 public class Precondition extends OptionalPrecondition
    implements Element {
```

```
    public String toString() {
        return "Precondition()";
    }


    public void accept(Visitor visitor) {
        visitor.visitPrecondition(this);
    }
}
```

E.3.38 PrefixExpr.java

```
package translator.rslast;

import translator.lib.*;

public abstract class PrefixExpr extends ValueExpr implements
    Element {

    public abstract String toString();

    public abstract void accept(Visitor visitor);


}
```

E.3.39 ProductTypeExpr.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;

public class ProductTypeExpr extends TypeExpr implements Element
    {
    private RSLList<TypeExpr> typeExprList;

    public ProductTypeExpr(RSLList<TypeExpr> typeExprList) {
        this.typeExprList = typeExprList;
    }

    public RSLList<TypeExpr> getTypeExprList() {
        return typeExprList;
    }

    public void setTypeExprList(RSLList<TypeExpr> typeExprList)
        {
        this.typeExprList = typeExprList;
    }

```

```
    public String toString() {
        StringBuffer result = new StringBuffer();
        result.append("ProductTypeExpr(");
        for(TypeExpr te : typeExprList.getList()) {
            result.append(te.toString());
            result.append(", ");
        }
        if(!typeExprList.getList().isEmpty())
            result.delete(result.length() - 2, result.length());
        result.append(")");
        return result.toString();
    }

    public void accept(Visitor visitor) {
        visitor.visitProductTypeExpr(this);
    }
}
```

E.3.40 RecordPattern.java

```
package translator.rslast;

import translator.rsllib.*;
import translator.lib.*;

public class RecordPattern
    extends RSLElement
    implements Pattern, TypeEvaluator {
  private ValueOrVariableName valueOrVariableName;
  private RSLList < Pattern > innerPatternList;

  public RecordPattern(ValueOrVariableName valueOrVariableName,
                       RSLList < Pattern > innerPatternList) {
    this.valueOrVariableName = valueOrVariableName;
    this.innerPatternList = innerPatternList;
  }

  public ValueOrVariableName getValueOrVariableName() {
    return valueOrVariableName;
  }

  public void setValueOrVariableName(ValueOrVariableName
     valueOrVariableName) {
    this.valueOrVariableName = valueOrVariableName;
  }

  public RSLList < Pattern > getInnerPatternList() {
    return innerPatternList;
  }
```

```
30    public void setInnerPatternList(RSLList < Pattern >
          innerPatternList) {
        this.innerPatternList = innerPatternList;
      }

      public String toString() {
35      StringBuffer result = new StringBuffer("RecordPattern(");
        result.append(valueOrVariableName.toString());
        result.append(", InnerPatternList(");
        for (Pattern p : innerPatternList.getList()) {
          result.append(p.toString());
40        result.append(", ");
        }
        if (!innerPatternList.getList().isEmpty()) {
          result.delete(result.length() - 2, result.length());
        }
45      result.append("))");
        return result.toString();
      }

      public void accept(Visitor visitor) {
50      visitor.visitRecordPattern(this);
      }
    }
```

E.3.41 RecordVariant.java

```
    package translator.rslast;

    import translator.lib.*;
    import translator.rsllib.*;
5
    public class RecordVariant
        extends Variant {
      private Constructor constructor;
      private RSLList < ComponentKind > componentKindList;
10
      public RecordVariant(Constructor constructor,
                           RSLList < ComponentKind >
                               componentKindList) {
        this.constructor = constructor;
        this.componentKindList = componentKindList;
15    }

      public Constructor getConstructor() {
        return constructor;
      }
20
```

```
  public void setConstructor(Constructor constructor) {
    this.constructor = constructor;
  }

  public RSLList < ComponentKind > getComponentKindList() {
    return this.componentKindList;
  }

  public void setComponentKindList(RSLList < ComponentKind >
      componentKindList) {
    this.componentKindList = componentKindList;
  }

  public String toString() {
    StringBuffer result = new StringBuffer("RecordVariant(");
    result.append(constructor.toString());
    result.append(",");
    result.append("ComponentKindList(");
    for (ComponentKind componentKind : componentKindList.getList
        ()) {
      result.append(componentKind.toString());
      result.append(", ");
    }
    if (!componentKindList.getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")");
    result.append(")");

    return result.toString();
  }

  public void accept(Visitor visitor) {
    visitor.visitRecordVariant(this);
  }
}
```

E.3.42 RSLAst.java

```
package translator.rslast;

import translator.lib.*;

public class RSLAst extends RSLElement {
    private LibModule module;

    public RSLAst(LibModule module) {
        this.module = module;
    }
```

```
    public LibModule getLibModule() {
        return module;
    }

    public void setLibModule(LibModule module) {
        this.module = module;
    }

    public String toString() {
        return "RSLAST(\n" + module.toString() + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitRSLAst(this);
    }
}
```

E.3.43 RSLInfixOp.java

```
package translator.rslast;

import translator.lib.*;

public class RSLInfixOp extends RSLOp {
    public static final RSLInfixOp RSL_INFIX_OP_PLUS = new
       RSLInfixOp("+");
    public static final RSLInfixOp RSL_INFIX_OP_EQUALS = new
       RSLInfixOp("=");
    public static final RSLInfixOp RSL_INFIX_OP_STAR = new
       RSLInfixOp("*");
    public static final RSLInfixOp RSL_INFIX_OP_HAT = new
       RSLInfixOp("^");

    private String text;

    private RSLInfixOp(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public String toString() {
        String result = "RSLInfixOp.";
        if(this == RSL_INFIX_OP_PLUS)
            result += "PLUS";
        else if(this == RSL_INFIX_OP_EQUALS)
```

```
            result += "EQUALS";
        else if (this == RSL_INFIX_OP_STAR)
            result += "STAR";
        else if (this == RSL_INFIX_OP_HAT)
            result += "HAT";
        else
            result += "UNKNOWN_OPERATOR";
        return result;
    }

    public void accept(Visitor visitor) {
        visitor.visitRSLInfixOp(this);
    }
}
```

E.3.44 RSLOp.java

```
package translator.rslast;

import translator.lib.*;

public abstract class RSLOp extends RSLElement {

    public abstract String toString();

    public abstract void accept(Visitor visitor);
}
```

E.3.45 RSLPrefixOp.java

```
package translator.rslast;

import translator.lib.*;

public class RSLPrefixOp extends RSLOp {
    public static final RSLPrefixOp RSL_PREFIX_OP_HD = new
        RSLPrefixOp("hd");
    public static final RSLPrefixOp RSL_PREFIX_OP_TL = new
        RSLPrefixOp("tl");

    private String text;

    private RSLPrefixOp(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }
```

```
    public String toString() {
        String result = "RSL_PREFIX_OP.";
        if(this == RSL_PREFIX_OP_HD)
            result += "HD";
        else if(this == RSL_PREFIX_OP_TL)
            result += "TL";
        else
            result += "UNKNOWN_OPERATOR";
        return result;
    }

    public void accept(Visitor visitor) {
        visitor.visitRSLPrefixOp(this);
    }
}
```

E.3.46 SchemeDef.java

```
package translator.rslast;

import translator.lib.*;

public class SchemeDef extends RSLElement {
    private ClassExpr classExpr;
    private Id id;

    public SchemeDef(Id id, ClassExpr classExpr) {
        this.id = id;
        this.classExpr = classExpr;
    }

    public ClassExpr getClassExpr() {
        return classExpr;
    }

    public void setClassExpr(ClassExpr classExpr) {
        this.classExpr = classExpr;
    }

    public Id getId() {
        return id;
    }

    public void setId(Id id) {
        this.id = id;
    }

    public String toString() {
```

```
        return "SchemeDef(\n" + id.toString() + ", " +
            classExpr.toString() + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitSchemeDef(this);
    }
}
```

E.3.47 SchemeInstantiation.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;

public class SchemeInstantiation extends ClassExpr implements
    Element {
    private Id id;

    public SchemeInstantiation(Id id) {
        this.id = id;
    }

    public Id getId() {
        return id;
    }

    public void setId(Id id) {
        this.id = id;
    }

    public String toString() {
        StringBuffer result = new StringBuffer("
            SchemeInstantiation(\n");
        result.append(id.toString());
        result.append(")");
        return result.toString();
    }

    public void accept(Visitor visitor) {
        visitor.visitSchemeInstantiation(this);
    }
}
```

E.3.48 ShortRecordDef.java

```
package translator.rslast;
```

```
import translator.lib.*;
import translator.rsllib.*;

public class ShortRecordDef extends TypeDef implements Element,
    TypeEvaluator {
    private Id id;
    private RSLList<ComponentKind> componentKindString;

    public ShortRecordDef(Id id, RSLList<ComponentKind>
        componentKindString) {
        this.id = id;
        this.componentKindString = componentKindString;
    }

    public Id getId() {
        return id;
    }

    public void setId(Id id) {
        this.id = id;
    }

    public RSLList<ComponentKind> getComponentKindString() {
        return this.componentKindString;
    }

    public void setVariantList(RSLList<ComponentKind>
        componentKindString) {
        this.componentKindString = componentKindString;
    }

    public String toString() {
        StringBuffer result = new StringBuffer("ShortRecordDef("
            );
        result.append(id.toString());
        result.append(",");
        result.append("ComponentKindString(");
        for(ComponentKind c : componentKindString.getList()) {
            result.append(c.toString());
            result.append(", ");
        }
        if(!componentKindString.getList().isEmpty())
            result.delete(result.length() - 2, result.length());
        result.append(")");
        result.append(")");
        return result.toString();
    }

    public void accept(Visitor visitor) {
```

```
        visitor.visitShortRecordDef(this);
    }
}
```

E.3.49 SingleTyping.java

```
package translator.rslast;

import translator.lib.*;

public class SingleTyping extends RSLElement {

    private Binding binding;
    private TypeExpr typeExpr;

    public SingleTyping(Binding binding, TypeExpr typeExpr) {
        this.binding = binding;
        this.typeExpr = typeExpr;

    }

    public Binding getBinding() {
        return binding;
    }

    public void setBinding(Binding binding) {
        this.binding = binding;
    }

    public TypeExpr getTypeExpr() {
        return typeExpr;
    }

    public void setTypeExpr(TypeExpr typeExpr) {
        this.typeExpr = typeExpr;
    }

    public String toString() {
        return "SingleTyping(" + binding.toString() + ", " +
            typeExpr.toString() + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitSingleTyping(this);
    }
}
```

E.3.50 SortDef.java

```
package translator.rslast;

import translator.lib.*;

public class SortDef extends TypeDef implements Element {
    private Id id;

    public SortDef(Id id) {
        this.id = id;
    }

    public Id getId() {
        return id;
    }

    public void setId(Id id) {
        this.id = id;
    }

    public String toString() {
        return "SortDef(" + id.toString() + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitSortDef(this);
    }
}
```

E.3.51 StructuredExpr.java

```
package translator.rslast;

import translator.lib.*;

public abstract class StructuredExpr extends ValueExpr
   implements Element {

    public abstract String toString();

    public abstract void accept(Visitor visitor);
}
```

E.3.52 TestDecl.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;

```

```
public class TestDecl extends Decl implements Element {
    private RSLList<TestDef> testDefList;

    public TestDecl(RSLList<TestDef> testDefList) {
        this.testDefList = testDefList;
    }

    public RSLList<TestDef> getTestDefList() {
        return testDefList;
    }

    public void setTestDefList(RSLList<TestDef> testDefList) {
        this.testDefList = testDefList;
    }

    public String toString() {
        StringBuffer result = new StringBuffer("TestDecl(");
        for(TestDef testDef : testDefList.getList()) {
            result.append(testDef.toString() + ", ");
        }
        if(!testDefList.getList().isEmpty())
            result.delete(result.length() - 2, result.length());
        result.append(")");
        return result.toString();
    }

    public void accept(Visitor visitor) {
        visitor.visitTestDecl(this);
    }
}
```

E.3.53 TestDef.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;

public class TestDef extends RSLElement {
    private Id id;
    private ValueExpr valueExpr;

    public TestDef(Id id, ValueExpr valueExpr) {
        this.id = id;
        this.valueExpr = valueExpr;
    }

    public Id getId() {
        return id;
```

```
    }

    public void setId(Id id) {
        this.id = id;
    }

    public ValueExpr getValueExpr() {
        return valueExpr;
    }

    public void setValueExpr(ValueExpr valueExpr) {
        this.valueExpr = valueExpr;
    }

    public String toString() {
        StringBuffer result = new StringBuffer("TestDef(");
        result.append(id.toString());
        result.append(",");
        result.append(valueExpr.toString());
        result.append(")");
        return result.toString();
    }

    public void accept(Visitor visitor) {
        visitor.visitTestDef(this);
    }
}
```

E.3.54 TypeDecl.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;

public class TypeDecl extends Decl implements Element {
    private RSLList<TypeDef> typeDefList;

    public TypeDecl(RSLList<TypeDef> typeDefList) {
        this.typeDefList = typeDefList;
    }

    public RSLList<TypeDef> getTypeDefList() {
        return typeDefList;
    }

    public void setTypeDefList(RSLList<TypeDef> typeDefList) {
        this.typeDefList = typeDefList;
    }
```

```
20
    public String toString() {
        StringBuffer result = new StringBuffer("TypeDecl(");
        for(TypeDef typeDef : typeDefList.getList()) {
            result.append(typeDef.toString() + ", ");
        }
        if(!typeDefList.getList().isEmpty())
            result.delete(result.length() - 2, result.length());
        result.append(")");
        return result.toString();
    }

    public void accept(Visitor visitor) {
        visitor.visitTypeDecl(this);
    }
}
```

E.3.55 TypeDef.java

```
package translator.rslast;

import translator.lib.*;

public abstract class TypeDef extends RSLElement implements
    Element,
    TypeEvaluator {

    public abstract String toString();
    public abstract void accept(Visitor visitor);
}
```

E.3.56 TypeExpr.java

```
package translator.rslast;

import translator.lib.*;

public abstract class TypeExpr extends RSLElement implements
    TypeEvaluator{

    public abstract String toString();
    public abstract void accept(Visitor visitor);
}
```

E.3.57 TypeLiteral.java

```
package translator.rslast;

import translator.lib.*;
```

```
5  public class TypeLiteral extends TypeExpr implements Element {
       public static final TypeLiteral RSLUNIT = new TypeLiteral("
           Unit");
       public static final TypeLiteral RSLBOOL = new TypeLiteral("
           Bool");
       public static final TypeLiteral RSLINT = new TypeLiteral("
           Int");
       public static final TypeLiteral RSLNAT = new TypeLiteral("
           Nat");
10     public static final TypeLiteral RSLREAL = new TypeLiteral("
           Real");
       public static final TypeLiteral RSLTEXT = new TypeLiteral("
           Text");
       public static final TypeLiteral RSLCHAR = new TypeLiteral("
           Char");

       private String text;
15
       private TypeLiteral(String text) {
           this.text = text;
       }

20     public String getText() {
           return text;
       }

       public String toString() {
25         String result = "TypeLiteral.";
           if(this == RSLUNIT)
               result += "RSLUNIT";
           else if(this == RSLBOOL)
               result += "RSLBOOL";
30         else if(this == RSLINT)
               result += "RSLINT";
           else if(this == RSLNAT)
               result += "RSLNAT";
           else if(this == RSLREAL)
35             result += "RSLREAL";
           else if(this == RSLTEXT)
               result += "RSLTEXT";
           else if(this == RSLCHAR)
               result += "RSLCHAR";
40         return result;
       }

       public void accept(Visitor visitor) {
           visitor.visitTypeLiteral(this);
45     }
```

```
}
```

E.3.58 TypeName.java

```
package translator.rslast;

import translator.lib.*;

public class TypeName extends TypeExpr {
    private Id id;

    public TypeName(Id id) {
        this.id = id;
    }

    public Id getId() {
        return id;
    }

    public void setId(Id id) {
        this.id = id;
    }

    public String toString() {
        return "TypeName(" + id.toString() + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitTypeName(this);
    }
}
```

E.3.59 ValueDecl.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;

public class ValueDecl extends Decl implements Element {
    private RSLList<ValueDef> valueDefList;

    public ValueDecl(RSLList<ValueDef> valueDefList) {
        this.valueDefList = valueDefList;
    }

    public RSLList<ValueDef> getValueDefList() {
        return valueDefList;
    }
```

```
    public void setValueDefList(RSLList<ValueDef> valueDefList)
        {
        this.valueDefList = valueDefList;
    }

    public String toString() {
        StringBuffer result = new StringBuffer("ValueDecl(\n");
        for(ValueDef valueDef : valueDefList.getList()) {
            result.append(valueDef.toString());
            result.append(", \n");
        }
        if(!valueDefList.getList().isEmpty())
            result.delete(result.length() - 3, result.length());
        result.append(")");
        return result.toString();
    }

    public void accept(Visitor visitor) {
        visitor.visitValueDecl(this);
    }
}
```

E.3.60 ValueDef.java

```
package translator.rslast;

import translator.lib.*;

public abstract class ValueDef extends RSLElement {

    public abstract String toString();
    public abstract void accept(Visitor visitor);
}
```

E.3.61 ValueExpr.java

```
package translator.rslast;

import translator.lib.*;

public abstract class ValueExpr extends RSLElement {

    public abstract String toString();

    public abstract void accept(Visitor visitor);
}
```

E.3.62 ValueInfixExpr.java

```
package translator.rslast;

import translator.lib.*;

public class ValueInfixExpr extends InfixExpr implements Element
    {
    private ValueExpr left;
    private RSLInfixOp op;
    private ValueExpr right;

    public ValueInfixExpr(ValueExpr left, RSLInfixOp op,
        ValueExpr right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public ValueExpr getLeft() {
        return left;
    }

    public void setLeft(ValueExpr left) {
        this.left = left;
    }

    public RSLInfixOp getOp() {
        return op;
    }

    public void setOp(RSLInfixOp op) {
        this.op = op;
    }

    public ValueExpr getRight() {
        return right;
    }

    public void setRight(ValueExpr right) {
        this.right = right;
    }

    public String toString() {
        StringBuffer result = new StringBuffer("ValueInfixExpr("
            );
        result.append(left.toString() + ", ");
        result.append(op.toString() + ", ");
        result.append(right.toString());
```

```
45          result.append(")");
            return result.toString();
        }

        public void accept(Visitor visitor) {
50          visitor.visitValueInfixExpr(this);
        }
    }
```

E.3.63 ValueLiteral.java

```
package translator.rslast;

import translator.lib.*;

5   public abstract class ValueLiteral extends ValueExpr {

        public abstract String toString();

        public abstract void accept(Visitor visitor);
10  }
```

E.3.64 ValueLiteralBool.java

```
package translator.rslast;

import translator.lib.*;

5   public class ValueLiteralBool extends ValueLiteral {
        private String s;

        public ValueLiteralBool(String s) {
            this.s = s;
10      }

        public String getText() {
            return s;
        }
15
        public String toString() {
            return "ValueLiteralBool(" + s + ")";
        }

20      public void accept(Visitor visitor) {
            visitor.visitValueLiteralBool(this);
        }
    }
```

E.3.65 ValueLiteralChar.java

```
package translator.rslast;

import translator.lib.*;

public class ValueLiteralChar extends ValueLiteral implements
    Element {
    private String s;

    public ValueLiteralChar(String s) {
        this.s = s;
    }

    public String getText() {
        return s;
    }

    public String toString() {
        return "ValueLiteralChar(" + s + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitValueLiteralChar(this);
    }
}
```

E.3.66 ValueLiteralInteger.java

```
package translator.rslast;

import translator.lib.*;

public class ValueLiteralInteger extends ValueLiteral implements
    Element {
    private String s;

    public ValueLiteralInteger(String s) {
        this.s = s;
    }

    public String getText() {
        return s;
    }

    public String toString() {
        return "ValueLiteralInteger(" + s + ")";
    }
```

```
20      public void accept(Visitor visitor) {
            visitor.visitValueLiteralInteger(this);
        }
    }
```

E.3.67 ValueLiteralPattern.java

```
package translator.rslast;

import translator.lib.*;

5   public class ValueLiteralPattern extends RSLElement implements
        Pattern, Element {
        private ValueLiteral valueLiteral;

        public ValueLiteralPattern(ValueLiteral valueLiteral) {
            this.valueLiteral = valueLiteral;
10      }

        public ValueLiteral getValueLiteral() {
            return valueLiteral;
        }
15
        public void setValueLiteral(ValueLiteral valueLiteral) {
            this.valueLiteral = valueLiteral;
        }

20      public String toString() {
            return "ValueLiteralPattern(" + valueLiteral + ")";
        }

        public void accept(Visitor visitor) {
25          visitor.visitValueLiteralPattern(this);
        }
    }
```

E.3.68 ValueLiteralReal.java

```
package translator.rslast;

import translator.lib.*;

5   public class ValueLiteralReal extends ValueLiteral {
        private String s;

        public ValueLiteralReal(String s) {
            this.s = s;
10      }
```

```
    public String getText() {
        return s;
    }

    public String toString() {
        return "ValueLiteralReal(" + s + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitValueLiteralReal(this);
    }
}
```

E.3.69 ValueLiteralText.java

```
package translator.rslast;

import translator.lib.*;

public class ValueLiteralText extends ValueLiteral implements
    Element {
    private String s;

    public ValueLiteralText(String s) {
        this.s = s;
    }

    public String getText() {
        return s;
    }

    public String toString() {
        return "ValueLiteralText(" + s + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitValueLiteralText(this);
    }
}
```

E.3.70 ValueOrVariableName.java

```
package translator.rslast;

import translator.lib.*;

public class ValueOrVariableName extends ValueExpr {
    private Id id;
```

```
    public ValueOrVariableName(Id id) {
        this.id = id;
    }

    public Id getId() {
        return id;
    }

    public void setId(Id id) {
        this.id = id;
    }

    public String toString() {
        return "ValueOrVariableName(" + id.toString() + ")";
    }

    public void accept(Visitor visitor) {
        visitor.visitValueOrVariableName(this);
    }
}
```

E.3.71 ValuePrefixExpr.java

```
package translator.rslast;

import translator.lib.*;

public class ValuePrefixExpr extends PrefixExpr implements
    Element {
    private RSLPrefixOp op;
    private ValueExpr valueExpr;

    public ValuePrefixExpr(RSLPrefixOp op, ValueExpr valueExpr)
        {
        this.op = op;
        this.valueExpr = valueExpr;
    }

    public RSLPrefixOp getOp() {
        return op;
    }

    public void setOp(RSLPrefixOp op) {
        this.op = op;
    }

    public ValueExpr getValueExpr() {
        return valueExpr;
    }
```

```
25
    public void setValueExpr(ValueExpr valueExpr) {
        this.valueExpr = valueExpr;
    }

30  public String toString() {
        StringBuffer result = new StringBuffer("ValuePrefixExpr(
            ");
        result.append(op.toString() + ", ");
        result.append(valueExpr.toString());
        result.append(")");
35      return result.toString();
    }


    public void accept(Visitor visitor) {
40      visitor.visitValuePrefixExpr(this);
    }
}
```

E.3.72 Variant.java

```
package translator.rslast;

import translator.lib.*;

5 public abstract class Variant extends RSLElement {
    public abstract String toString();

    public abstract void accept(Visitor visitor);
}
```

E.3.73 VariantDef.java

```
package translator.rslast;

import translator.lib.*;
import translator.rsllib.*;
5
public class VariantDef extends TypeDef implements Element {
    private Id id;
    private RSLList<Variant> variantList;

10  public VariantDef(Id id, RSLList<Variant> variantList) {
        this.id = id;
        this.variantList = variantList;
    }

15  public Id getId() {
```

```
        return id;
    }

    public void setId(Id id) {
        this.id = id;
    }

    public RSLList<Variant> getVariantList() {
        return this.variantList;
    }

    public void setVariantList(RSLList<Variant> variantList) {
        this.variantList = variantList;
    }

    public String toString() {
        StringBuffer result = new StringBuffer("VariantDef(");
        result.append(id.toString());
        result.append(",");
        result.append("VariantList(");
        for(Variant v : variantList.getList()) {
            result.append(v.toString());
            result.append(", ");
        }
        if(!variantList.getList().isEmpty())
            result.delete(result.length() - 2, result.length());
        result.append(")");
        result.append(")");
        return result.toString();
    }

    public void accept(Visitor visitor) {
        visitor.visitVariantDef(this);
    }
}
```

E.3.74 WildcardPattern.java

```
package translator.rslast;

import translator.lib.*;

public class WildcardPattern extends RSLElement implements
    Pattern, Element {

    public WildcardPattern() { }

    public String toString() {
        return "WildcardPattern()";
```

```
    }

    public void accept(Visitor visitor) {
        visitor.visitWildcardPattern(this);
    }
}
```

E.4 translator.javaast

E.4.1 ArrayCreation.java

```
package translator.javaast;

import translator.lib.*;
import translator.rsllib.*;

public class ArrayCreation
    extends Expression
    implements JavaElement {

  private JavaType type;
  private Expression optionalExpression;
  private RSLList < Expression > optionalExpressionList;

  public ArrayCreation(JavaType type, Expression
      optionalExpression,
                       RSLList < Expression >
                          optionalExpressionList) {
    this.type = type;
    this.optionalExpression = optionalExpression;
    this.optionalExpressionList = optionalExpressionList;
  }

  public JavaType getType() {
    return type;
  }

  public void setType(JavaType type) {
    this.type = type;
    ;
  }

  public Expression getExpression() {
    return optionalExpression;
  }

  public void setExpression(Expression optionalExpression) {
    this.optionalExpression = optionalExpression;
```

```
  }

  public RSLList < Expression > getExpressionList() {
    return optionalExpressionList;
  }

  public void setExpression(RSLList < Expression >
      optionalExpressionList) {
    this.optionalExpressionList = optionalExpressionList;
  }

  public String toString() {
    StringBuffer result = new StringBuffer("ArrayCreation(" +
        type.toString() +
                                           ", ");
    if (optionalExpression != null) {
      result.append(optionalExpression.toString());
    }
    else {
      result.append("noExpression()");
    }
    result.append(", ");
    if (optionalExpressionList != null) {
      result.append("ExpressionList(");
      for (Expression expression : optionalExpressionList.
          getList()) {
        result.append(expression.toString());
        result.append(", ");
      }
      if (!optionalExpressionList.getList().isEmpty()) {
        result.delete(result.length() - 2, result.length());
      }
      result.append(")");
    }
    else {
      result.append("NoExpressionList()");
    }
    result.append(")");
    return result.toString();
  }

  public void accept(JavaVisitor visitor) {
    visitor.visitArrayCreation(this);
  }
}
```

E.4.2 ArrayType.java

```
package translator.javaast;
```

```
import translator.lib.*;

public class ArrayType extends JavaType implements JavaElement {
    private JavaType type;

    public ArrayType(JavaType type) {
        this.type = type;
    }

    public JavaType getType() {
        return type;
    }

    public void setType(JavaType type) {
        this.type = type;
    }

    public String toString() {
        return "ArrayType(" + type.toString() + ")";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitArrayType(this);
    }
}
```

E.4.3 AssignmentExpression.java

```
package translator.javaast;

import translator.lib.*;

public class AssignmentExpression
    extends Expression
    implements JavaElement {
  private Expression leftHandSide;
  private AssignmentOperator op;
  private Expression rightHandSide;

  public AssignmentExpression(Expression leftHandSide,
     AssignmentOperator op,
                              Expression rightHandSide) {
    this.leftHandSide = leftHandSide;
    this.op = op;
    this.rightHandSide = rightHandSide;
  }

  public Expression getLeftHandSide() {
```

```
20      return leftHandSide;
    }

    public void setLeftHandSide(Expression leftHandSide) {
      this.leftHandSide = leftHandSide;
25  }

    public AssignmentOperator getOp() {
      return op;
    }
30
    public void setOp(AssignmentOperator op) {
      this.op = op;
    }

35  public Expression getRightHandSide() {
      return rightHandSide;
    }

    public void setRightHandSide(Expression rightHandSide) {
40    this.rightHandSide = rightHandSide;
    }

    public String toString() {
      String result = "AssignmentExpression(" +
45        (leftHandSide != null ? leftHandSide.toString() : "
            UNEXPECTED NULL") +
        ", ";
      result += (op != null ? op.toString() : "UNEXPECTED NULL")
        + ", ";
      result +=
        (rightHandSide != null ? rightHandSide.toString() : "
            UNEXPECTED NULL") +
50      ")";

      return result;
    }

55  public void accept(JavaVisitor visitor) {
      visitor.visitAssignmentExpression(this);
    }
  }
```

E.4.4 AssignmentOperator.java

```
package translator.javaast;

import translator.lib.*;
```

```
public class AssignmentOperator implements JavaElement {
    public static final AssignmentOperator
        JAVA_ASSIGNMENT_OP_EQUAL =
         new AssignmentOperator("=");

    private String text;

    private AssignmentOperator(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public String toString() {
        String result = "AssignmentOperator.";
        if(this == JAVA_ASSIGNMENT_OP_EQUAL) {
            result += "JAVA_ASSIGNMENT_OP_EQUAL";
        }
        return result;
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitAssignmentOperator(this);
    }
}
```

E.4.5 Block.java

```
package translator.javaast;

import translator.lib.*;
import translator.rsllib.*;

public class Block implements JavaElement {

    private RSLList<Statement> statementList;

    public Block(RSLList<Statement> statementList) {
        this.statementList = statementList;
    }

    public RSLList<Statement> getStatementList() {
        return statementList;
    }

    public void setStatementList(RSLList<Statement>
        statementList) {
```

```
        this.statementList = statementList;
    }

    public String toString() {
        StringBuffer result = new StringBuffer();
        result.append("Block(");
        result.append("\nStatementList(");
        for(Statement s : statementList.getList()) {
            result.append(s.toString());
            result.append(",\n");
        }
        if(!statementList.getList().isEmpty())
            result.delete(result.length() - 2, result.length());
        result.append("))\n");
        return result.toString();
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitBlock(this);
    }
}
```

E.4.6 BooleanLiteral.java

```
package translator.javaast;

import translator.lib.*;

public class BooleanLiteral extends Expression implements
    JavaElement {
    private String text;

    public BooleanLiteral(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    public String toString() {
        return "BooleanLiteral(" + text + ")";
    }

    public void accept(JavaVisitor visitor) {
```

```
25          visitor.visitBooleanLiteral(this);
        }
    }
```

E.4.7 CastExpression.java

```
package translator.javaast;

import translator.lib.*;

5   public class CastExpression extends Expression implements
        JavaElement {
        private Expression expression;
        private JavaType type;

        public CastExpression(JavaType type, Expression expression)
            {
10          this.type = type;
            this.expression = expression;
        }

        public JavaType getType() {
15          return type;
        }

        public void setType(JavaType type) {
            this.type = type;;
20      }


        public Expression getExpression() {
            return expression;
25      }

        public void setExpression(Expression expressionm) {
            this.expression = expression;
        }
30
        public String toString() {
            return "CastExpression(" + type.toString() + ", " +
                expression.toString() + ")";
        }
35
        public void accept(JavaVisitor visitor) {
            visitor.visitCastExpression(this);
        }
    }
```

E.4.8 CharLiteral.java

```
package translator.javaast;

import translator.lib.*;

public class CharLiteral extends Expression implements
    JavaElement {
    private String text;

    public CharLiteral(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    public String toString() {
        return "CharLiteral(" + text + ")";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitCharLiteral(this);
    }
}
```

E.4.9 ClassDeclaration.java

```
package translator.javaast;

import translator.lib.*;
import translator.rsllib.*;

public class ClassDeclaration
    extends TypeDeclaration
    implements JavaElement {
  private RSLList < Modifier > modifierList;
  private SimpleName name;
  private SimpleName optionalExtend;
  private RSLList < SimpleName > implementList;
  private RSLList < ConstructorDeclaration >
     constructorDeclarationList;
  private RSLList < MethodDeclaration > methodDeclarationList;
  private RSLList < FieldDeclaration > fieldDeclarationList;
```

```
  private RSLList < ClassDeclaration > classDeclarationList;

  public ClassDeclaration(RSLList < Modifier > modifierList ,
      SimpleName name,
                          SimpleName optionalExtend ,
                          RSLList < SimpleName > implementList ,
                          RSLList < ConstructorDeclaration >
                          constructorDeclarationList ,
                          RSLList < MethodDeclaration >
                              methodDeclarationList ,
                          RSLList < FieldDeclaration >
                              fieldDeclarationList ,
                          RSLList < ClassDeclaration >
                              classDeclarationList) {
    this.modifierList = modifierList;
    this.name = name;
    this.optionalExtend = optionalExtend;
    this.implementList = implementList;
    this.constructorDeclarationList = constructorDeclarationList
        ;
    this.methodDeclarationList = methodDeclarationList;
    this.fieldDeclarationList = fieldDeclarationList;
    this.classDeclarationList = classDeclarationList;
  }

  public RSLList < Modifier > getModifierList() {
    return modifierList;
  }

  public void setModifierList(RSLList < Modifier > modifierList)
      {
    this.modifierList = modifierList;
  }

  public SimpleName getSimpleName() {
    return name;
  }

  public void setSimpleName(SimpleName name) {
    this.name = name;
  }

  public SimpleName getOptionalExtend() {
    return optionalExtend;
  }

  public RSLList < SimpleName > getImplementList() {
    return implementList;
  }
```

```
60    public void setImplementList(RSLList < SimpleName >
          implementList) {
        this.implementList = implementList;
      }

      public RSLList < ConstructorDeclaration >
          getConstructorDeclarationList() {
65      return constructorDeclarationList;
      }

      public void setConstructorDeclarationList(RSLList <
          ConstructorDeclaration >
                                                  constructorDeclarationList
                                                      ) {
70      this.constructorDeclarationList = constructorDeclarationList
          ;
      }

      public RSLList < MethodDeclaration > getMethodDeclarationList
          () {
        return methodDeclarationList;
75    }

      public void setMethodDeclarationList(RSLList <
          MethodDeclaration >
                                           methodDeclarationList) {
        this.methodDeclarationList = methodDeclarationList;
80    }

      public RSLList < FieldDeclaration > getFieldDeclarationList()
          {
        return fieldDeclarationList;
      }
85
      public void setFieldDeclarationList(RSLList < FieldDeclaration
          >
                                          fieldDeclaration) {
        this.fieldDeclarationList = fieldDeclarationList;
      }
90
      public RSLList < ClassDeclaration > getClassDeclarationList()
          {
        return classDeclarationList;
      }

95    public void setClassDeclarationList(RSLList < ClassDeclaration
          >
                                          classDeclarationList) {
```

```
    this.classDeclarationList = classDeclarationList;
  }

  public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("TypeDeclaration(");
    result.append("ModifierList(");
    for (Modifier m : modifierList.getList()) {
      result.append(m.toString());
      result.append(", ");
    }
    if (!modifierList.getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append("),");
    result.append(name.toString());
    result.append(",\n");
    result.append("MethodDeclarationList(");
    for (MethodDeclaration md : methodDeclarationList.getList())
        {
      result.append(md.toString());
      result.append(", ");
    }
    if (!methodDeclarationList.getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append("),\n");
    result.append("FieldDeclarationList(");
    for (FieldDeclaration fd : fieldDeclarationList.getList()) {
      result.append(fd.toString());
      result.append(", ");
    }
    if (!fieldDeclarationList.getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")\n");
    result.append("ClassDeclarationList(");
    for (ClassDeclaration cd : classDeclarationList.getList()) {
      result.append(cd.toString());
      result.append(", ");
    }
    if (!classDeclarationList.getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")\n");
    result.append(")");

    return result.toString();
  }
```

```
145
    public void accept(JavaVisitor visitor) {
      visitor.visitClassDeclaration(this);
    }
  }
```

E.4.10 ClassInstanceCreation.java

```
  package translator.javaast;

  import translator.lib.*;
  import translator.rsllib.*;
5
  public class ClassInstanceCreation
      extends Expression
      implements JavaElement {
    private Expression expression;
10  private ReferenceType type;
    private RSLList < Expression > argumentList;

    public ClassInstanceCreation(Expression expression,
        ReferenceType type,
                                 RSLList < Expression >
                                    argumentList) {
15    this.expression = expression;
      this.type = type;
      this.argumentList = argumentList;
    }

20  public Expression getExpression() {
      return expression;
    }

    public void setExpression(Expression expression) {
25    this.expression = expression;
    }

    public RSLList < Expression > getArgumentList() {
      return argumentList;
30  }

    public void setArgumentList(RSLList < Expression >
       argumentList) {
      this.argumentList = argumentList;
    }
35
    public ReferenceType getType() {
      return type;
    }
```

```
40    public void setType(ReferenceType type) {
        this.type = type;
      }

      public String toString() {
45      StringBuffer result = new StringBuffer();
        result.append("ClassInstanceCreation(");
        if (expression != null) {
          result.append(expression.toString());
        }
50      else {
          result.append("noExpression()");
        }
        result.append(", ");
        result.append(type.toString());
55      result.append(", ExpressionList(");
        for (Expression e : argumentList.getList()) {
          result.append(e.toString());
          result.append(", ");
        }
60      result.append(")");
        result.append(")");
        return result.toString();
      }

65    public void accept(JavaVisitor visitor) {
        visitor.visitClassInstanceCreation(this);
      }
    }
```

E.4.11 CompilationUnit.java

```
package translator.javaast;

import translator.lib.*;
import translator.rsllib.*;
5
public class CompilationUnit
    implements JavaElement {

  private OptionalPackageDeclaration optionalPackageDeclaration;
10 private RSLList<ImportDeclaration> importDeclarationList;
  private RSLList<TypeDeclaration> typeDeclarationList;

  public CompilationUnit(OptionalPackageDeclaration
      optionalPackageDeclaration,
                         RSLList<ImportDeclaration>
                           importDeclarationList,
```

```
15                          RSLList < TypeDeclaration >
                               typeDeclarationList ) {
    this.optionalPackageDeclaration = optionalPackageDeclaration
        ;
    this.importDeclarationList = importDeclarationList;
    this.typeDeclarationList = typeDeclarationList;
  }
20
  public OptionalPackageDeclaration
      getOptionalPackageDeclaration() {
    return optionalPackageDeclaration;
  }

25 public void setOptionalPackageDeclaration(
      OptionalPackageDeclaration
                                              optionalPackageDeclaration
                                                 ) {
    this.optionalPackageDeclaration = optionalPackageDeclaration
        ;
  }

30 public RSLList < ImportDeclaration > getImportDeclarationList
      () {
    return importDeclarationList;
  }

  public RSLList < TypeDeclaration > getTypeDeclarationList() {
35  return typeDeclarationList;
  }

  public void setTypeDeclarationList(RSLList < TypeDeclaration >
                                       typeDeclarationList) {
40  this.typeDeclarationList = typeDeclarationList;
  }

  public void setImportDeclarationList(RSLList <
      ImportDeclaration >
                                         importDeclarationList) {
45  this.importDeclarationList = importDeclarationList;
  }

  public String toString() {
    StringBuffer result = new StringBuffer("CompilationUnit(");
50  if (optionalPackageDeclaration != null) {
      result.append(optionalPackageDeclaration.toString());
    }
    else {
      result.append("NoPackageDeclaration()");
55  }
```

```
      result.append(",\nImportDeclarationList(");
      for (ImportDeclaration id : importDeclarationList.getList())
          {
        result.append(id.toString());
        result.append(", ");
      }
      if (!importDeclarationList.getList().isEmpty()) {
        result.delete(result.length() - 2, result.length());
      }
      result.append(")");
      result.append(",\nTypeDeclarationList(");
      for (TypeDeclaration td : typeDeclarationList.getList()) {
        result.append(td.toString());
        result.append(", ");
      }
      if (!typeDeclarationList.getList().isEmpty()) {
        result.delete(result.length() - 2, result.length());
      }
      result.append(")\n");

      result.append(")\n");
      return result.toString();
    }

    public void accept(JavaVisitor visitor) {
      visitor.visitCompilationUnit(this);
    }
  }
```

E.4.12 ConstructorDeclaration.java

```
package translator.javaast;

import translator.lib.*;
import translator.rsllib.*;

public class ConstructorDeclaration
    implements JavaElement {
  private RSLList < Modifier > modifierList;
  private SimpleName name;
  private RSLList < SingleVariableDeclaration > argumentList;
  private Block block;

  public ConstructorDeclaration(RSLList < Modifier >
      modifierList,
                                SimpleName name,
                                RSLList <
                                    SingleVariableDeclaration >
                                argumentList, Block block) {
```

```
    this.modifierList = modifierList;
    this.name = name;
    this.argumentList = argumentList;
    this.block = block;
  }

  public RSLList < Modifier > getModifierList() {
    return modifierList;
  }

  public void setModifierList(RSLList < Modifier > modifierList)
      {
    this.modifierList = modifierList;
  }

  public SimpleName getSimpleName() {
    return name;
  }

  public void setSimpleName(SimpleName simpleName) {
    this.name = name;
  }

  public RSLList < SingleVariableDeclaration > getArgumentList()
      {
    return argumentList;
  }

  public void setArgumentList(RSLList <
     SingleVariableDeclaration >
                              argumentList) {
    this.argumentList = argumentList;
  }

  public Block getBlock() {
    return block;
  }

  public void setBlock(Block block) {
    this.block = block;
  }

  public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("\nMethodDeclaration(");
    result.append("ModifierList(");
    for (Modifier m : modifierList.getList()) {
      result.append(m.toString());
      result.append(", ");
```

```
      }
      if (!modifierList.getList().isEmpty()) {
        result.delete(result.length() - 2, result.length());
      }
      result.append(")\n");
      result.append(", ");
      result.append(name.toString());
      result.append(", ArgumentList(");
      for (SingleVariableDeclaration svd : argumentList.getList())
          {
        result.append(svd.toString());
        result.append(", ");
      }
      if (!argumentList.getList().isEmpty()) {
        result.delete(result.length() - 2, result.length());
      }
      result.append("),\n");
      if (block != null) {
        result.append(block.toString());
      }
      else {
        result.append("noBlock()");
      }
      result.append(")\n");

      return result.toString();
    }

    public void accept(JavaVisitor visitor) {
      visitor.visitConstructorDeclaration(this);
    }
  }
```

E.4.13 DoubleLiteral.java

```
package translator.javaast;

import translator.lib.*;

public class DoubleLiteral extends Expression implements
    JavaElement {
    private String text;

    public DoubleLiteral(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
```

```
    }

    public void setText(String text) {
        this.text = text;
    }

    public String toString() {
        return "DoubleLiteral(" + text + ")";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitDoubleLiteral(this);
    }
}
```

E.4.14 Expression.java

```
package translator.javaast;

import translator.lib.*;

public abstract class Expression implements JavaElement {

    public abstract String toString();

    public abstract void accept(JavaVisitor visitor);
}
```

E.4.15 ExpressionStatement.java

```
package translator.javaast;

import translator.lib.*;

public class ExpressionStatement extends Statement implements
    JavaElement {
    private Expression expression;

    public ExpressionStatement(Expression expression) {
        this.expression = expression;
    }

    public Expression getExpression() {
        return expression;
    }

    public void setExpression(Expression expression) {
        this.expression = expression;
    }
```

```
    public String toString() {
        return "ExpressionStatement(" + expression.toString() +
            ")";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitExpressionStatement(this);
    }
}
```

E.4.16 FieldAccessExpression.java

```
package translator.javaast;

import translator.lib.*;

public class FieldAccessExpression
    extends Expression
    implements JavaElement {
  private Expression expression;
  private SimpleName simpleName;

  public FieldAccessExpression(Expression expression, SimpleName
      simpleName) {
    this.expression = expression;
    this.simpleName = simpleName;
  }

  public Expression getExpression() {
    return expression;
  }

  public void setExpression(Expression expressionm) {
    this.expression = expression;
  }

  public SimpleName getSimpleName() {
    return simpleName;
  }

  public void setSimpleName(SimpleName simpleName) {
    this.simpleName = simpleName;
  }

  public String toString() {
    return "FieldAccessExpression(" + expression != null ?
        expression.toString() :
        "noExpression()" + ", " + simpleName.toString() + ")";
```

```
35    }

      public void accept(JavaVisitor visitor) {
        visitor.visitFieldAccessExpression(this);
      }
40  }
```

E.4.17 FieldDeclaration.java

```
package translator.javaast;

import translator.lib.*;
import translator.rsllib.*;
5
public class FieldDeclaration
    implements JavaElement {
  private RSLList < Modifier > modifierList;
  private JavaType type;
10  private VariableDeclarationFragment
      variableDeclarationFragment;

  public FieldDeclaration(RSLList < Modifier > modifierList,
      JavaType type,
                          VariableDeclarationFragment
                          variableDeclarationFragment) {
15    this.modifierList = modifierList;
    this.type = type;
    this.variableDeclarationFragment =
       variableDeclarationFragment;
  }

20  public RSLList < Modifier > getModifierList() {
    return modifierList;
  }

  public void setModifierList(RSLList < Modifier > modifierList)
      {
25    this.modifierList = modifierList;
  }

  public JavaType getType() {
    return type;
30  }

  public void setType(JavaType type) {
    this.type = type;
  }
35
  public VariableDeclarationFragment
```

```
      getVariableDeclarationFragment() {
    return variableDeclarationFragment;
  }

  public void setVariableDeclarationFragment(
      VariableDeclarationFragment
                                          variableDeclarationFragment
                                              ) {
    this.variableDeclarationFragment =
        variableDeclarationFragment;
  }

  public String toString() {
    StringBuffer result = new StringBuffer("FieldDeclaration(
        ModifierList(");
    for (Modifier m : modifierList.getList()) {
      result.append(m);
      result.append(", ");
    }
    if (!modifierList.getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append("), ");
    result.append(type.toString());
    result.append(", ");
    result.append(variableDeclarationFragment);
    result.append(")");
    return result.toString();
  }

  public void accept(JavaVisitor visitor) {
    visitor.visitFieldDeclaration(this);
  }
}
```

E.4.18 IfStatement.java

```
package translator.javaast;

import translator.lib.*;

public class IfStatement
    extends Statement
    implements JavaElement {
  private Expression condition;
  private Block ifBlock;
  private Block optionalElseBlock;

  public IfStatement(Expression condition, Block ifBlock,
```

```
                    Block optionalElseBlock) {
    this.condition = condition;
    this.ifBlock = ifBlock;
    this.optionalElseBlock = optionalElseBlock;
  }

  public Expression getCondition() {
    return condition;
  }

  public void setCondition(Expression condition) {
    this.condition = condition;
  }

  public Block getIfBlock() {
    return ifBlock;
  }

  public void setIfBlock(Block ifBlock) {
    this.ifBlock = ifBlock;
  }

  public Block getOptionalElseBlock() {
    return optionalElseBlock;
  }

  public IfStatement setOptionalElseBlock(Block
     optionalElseBlock) {
    this.optionalElseBlock = optionalElseBlock;
    return this;
  }

  public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("IfStatement(");
    result.append(condition.toString());
    result.append(", ");
    if (ifBlock != null) {
      result.append(ifBlock.toString());
    }
    else {
      result.append("UNEXPECTED NULL");
    }
    result.append(", ");
    if (optionalElseBlock != null) {
      result.append(optionalElseBlock.toString());
    }
    else {
      result.append("noOptionalElseBlock()");
```

```
    }
    result.append(")");
    return result.toString();
  }

  public void accept(JavaVisitor visitor) {
    visitor.visitIfStatement(this);
  }

}
```

E.4.19 ImportDeclaration.java

```
package translator.javaast;

import translator.lib.*;

public class ImportDeclaration implements JavaElement {
    private JavaName name;

    public ImportDeclaration(JavaName name) {
        this.name = name;
    }

    public JavaName getName() {
        return name;
    }

    public void setName(JavaName name) {
        this.name = name;
    }

    public String toString() {
        return "ImportDeclaration(" + name.toString() + ")";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitImportDeclaration(this);
    }
}
```

E.4.20 InfixExpression.java

```
package translator.javaast;

import translator.lib.*;

public class InfixExpression
    extends Expression
```

```
    implements JavaElement {
  private Expression left;
  private InfixOperator op;
  private Expression right;

  public InfixExpression(Expression left, InfixOperator op,
      Expression right) {
    this.left = left;
    this.op = op;
    this.right = right;
  }

  public Expression getLeft() {
    return left;
  }

  public void setLeft(Expression left) {
    this.left = left;
  }

  public InfixOperator getOp() {
    return op;
  }

  public void setOp(InfixOperator op) {
    this.op = op;
  }

  public Expression getRight() {
    return right;
  }

  public void setRight(Expression right) {
    this.right = right;
  }

  public String toString() {
    return "InfixExpression(" + left.toString() + ", " + op.
        toString() + ", " +
        right.toString() + ")";
  }

  public void accept(JavaVisitor visitor) {
    visitor.visitInfixExpression(this);
  }
}
```

E.4.21 InfixOperator.java

```
package translator.javaast;

import translator.lib.*;

public class InfixOperator
    implements JavaElement {
  public static final InfixOperator JAVA_INFIX_OP_PLUS = new
      InfixOperator("+");
  public static final InfixOperator JAVA_INFIX_OP_STAR = new
      InfixOperator("*");
  public static final InfixOperator JAVA_INFIX_OP_EQUALS = new
      InfixOperator(
      "==");

  private String text;

  private InfixOperator(String text) {
    this.text = text;
  }

  public String getText() {
    return text;
  }

  public String toString() {
    String result = "InfixOperator.";
    if (this == JAVA_INFIX_OP_PLUS) {
      result += "JAVA_INFIX_OP_PLUS";
    }
    else if (this == JAVA_INFIX_OP_STAR) {
      result += "JAVA_INFIX_OP_STAR";
    }
    else if (this == JAVA_INFIX_OP_EQUALS) {
      result += "JAVA_INFIX_OP_EQUALS";
    }
    else {
      result += "UNKNOWN";

    }
    return result;
  }

  public void accept(JavaVisitor visitor) {
    visitor.visitInfixOperator(this);
  }
}
```

E.4.22 InstanceOfExpression.java

```
package translator.javaast;

import translator.lib.*;

public class InstanceOfExpression
    extends Expression
    implements JavaElement {
  private Expression expression;
  private JavaType type;

  public InstanceOfExpression(Expression expression, JavaType
      type) {
    this.expression = expression;
    this.type = type;
  }

  public Expression getExpression() {
    return expression;
  }

  public void setExpression(Expression expressionm) {
    this.expression = expression;
  }

  public JavaType getType() {
    return type;
  }

  public void setType(JavaType type) {
    this.type = type;
    ;
  }

  public String toString() {
    return "InstanceOfExpression(" + expression.toString() + ",
        " +
         type.toString() + ")";
  }

  public void accept(JavaVisitor visitor) {
    visitor.visitInstanceOfExpression(this);
  }
}
```

E.4.23 IntegerLiteral.java

```
package translator.javaast;

import translator.lib.*;
```

```
5  public class IntegerLiteral extends Expression implements
       JavaElement {
       private String text;

       public IntegerLiteral(String text) {
           this.text = text;
10     }

       public String getText() {
           return text;
       }
15
       public void setText(String text) {
           this.text = text;
       }

20     public String toString() {
           return "IntegerLiteral(" + text + ")";
       }

       public void accept(JavaVisitor visitor) {
25         visitor.visitIntegerLiteral(this);
       }
   }
```

E.4.24 JavaAst.java

```
   package translator.javaast;

   import translator.lib.*;
   import translator.rsllib.*;
5
   public class JavaAst
       implements JavaElement {
     private RSLMap < String, CompilationUnit > compilationUnitMap;

10   public JavaAst(RSLMap < String, CompilationUnit >
         compilationUnitMap) {
       this.compilationUnitMap = compilationUnitMap;
     }

     public RSLMap < String, CompilationUnit >
         getCompilationUnitMap() {
15     return compilationUnitMap;
     }

     public void setCompilationUnitMap(RSLMap < String,
                                       CompilationUnit >
```

```
                                                compilationUnitMap) {
20      this.compilationUnitMap = compilationUnitMap;
    }

    public String toString() {
      StringBuffer result = new StringBuffer();
25      result.append("JavaAst(String->CompilationUnit(\n");
      for (String key : compilationUnitMap.getMap().keySet()) {
        result.append(key);
        result.append("->\n");
        result.append(compilationUnitMap.get(key).toString());
30        result.append(",\n");
      }
      if (!compilationUnitMap.getMap().isEmpty()) {
        result.delete(result.length() - 2, result.length());
      }
35      result.append("))");

      return result.toString();
    }

40  public void accept(JavaVisitor visitor) {
      visitor.visitJavaAst(this);
    }
}
```

E.4.25 JavaName.java

```
package translator.javaast;

import translator.lib.*;

5 public abstract class JavaName extends Expression implements
    JavaElement {

    public abstract String toString();

    public abstract void accept(JavaVisitor visitor);
10 }
```

E.4.26 JavaType.java

```
package translator.javaast;

import translator.lib.*;

5 public abstract class JavaType implements JavaElement {

    public abstract String toString();
```

```
    public abstract void accept(JavaVisitor visitor);
}
```

E.4.27 MethodDeclaration.java

```
package translator.javaast;

import translator.lib.*;
import translator.rsllib.*;

public class MethodDeclaration
    implements JavaElement {
  private RSLList < Modifier > modifierList;
  private SimpleName name;
  private JavaType returnType;
  private RSLList < SingleVariableDeclaration > argumentList;
  private Block block;

  public MethodDeclaration(RSLList < Modifier > modifierList,
      SimpleName name,
                           JavaType returnType,
                           RSLList < SingleVariableDeclaration
                               > argumentList,
                           Block block) {
    this.modifierList = modifierList;
    this.name = name;
    this.returnType = returnType;
    this.argumentList = argumentList;
    this.block = block;
  }

  public RSLList < Modifier > getModifierList() {
    return modifierList;
  }

  public void setModifierList(RSLList < Modifier > modifierList)
      {
    this.modifierList = modifierList;
  }

  public SimpleName getSimpleName() {
    return name;
  }

  public void setSimpleName(SimpleName simpleName) {
    this.name = name;
  }

```

```
  public JavaType getReturnType() {
    return returnType;
  }

  public void setReturnType(JavaType returnType) {
    this.returnType = returnType;
  }

  public RSLList < SingleVariableDeclaration > getArgumentList()
      {
    return argumentList;
  }

  public void setArgumentList(RSLList <
      SingleVariableDeclaration >
                              argumentList) {
    this.argumentList = argumentList;
  }

  public Block getBlock() {
    return block;
  }

  public void setBlock(Block block) {
    this.block = block;
  }

  public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("\nMethodDeclaration(");
    result.append("ModifierList(");
    for (Modifier m : modifierList.getList()) {
      result.append(m.toString());
      result.append(", ");
    }
    if (!modifierList.getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")\n");
    result.append(", ");
    result.append(name.toString());
    result.append(", ");
    result.append(returnType);
    result.append(", ArgumentList(");
    for (SingleVariableDeclaration svd : argumentList.getList())
        {
      result.append(svd.toString());
      result.append(", ");
    }
```

```
      if (!argumentList.getList().isEmpty()) {
        result.delete(result.length() - 2, result.length());
      }
      result.append("),\n");
      if (block != null) {
        result.append(block.toString());
      }
      else {
        result.append("noBlock()");
      }
      result.append(")\n");

      return result.toString();
    }

    public void accept(JavaVisitor visitor) {
      visitor.visitMethodDeclaration(this);
    }
}
```

E.4.28 MethodInvocation.java

```
package translator.javaast;

import translator.lib.*;
import translator.rsllib.*;

public class MethodInvocation
    extends Expression
    implements JavaElement {
  private Expression optionalExpression;
  private SimpleName name;
  private RSLList < Expression > argumentList;

  public MethodInvocation(Expression optionalExpression,
     SimpleName name,
                          RSLList < Expression > argumentList) {
    this.optionalExpression = optionalExpression;
    this.name = name;
    this.argumentList = argumentList;
  }

  public Expression getOptionalExpression() {
    return optionalExpression;
  }

  public void setOptionalExpression(Expression
     optionalExpression) {
    this.optionalExpression = optionalExpression;
```

```
  }

  public SimpleName getName() {
    return name;
  }

  public void setName(SimpleName name) {
    this.name = name;
  }

  public RSLList < Expression > getArgumentList() {
    return argumentList;
  }

  public void setArgumentList(RSLList < Expression >
      argumentList) {
    this.argumentList = argumentList;
  }

  public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("MethodInvocation(");
    if (optionalExpression != null) {
      result.append(optionalExpression.toString());
    }
    else {
      result.append("NoOptionalExpression()");
    }
    result.append(", ");
    result.append(name.toString());
    result.append(", ");
    result.append("ArgumentList(");
    for (Expression e : argumentList.getList()) {
      result.append(e.toString());
      result.append(", ");
    }
    if (!argumentList.getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")");
    result.append(")");

    return result.toString();
  }

  public void accept(JavaVisitor visitor) {
    visitor.visitMethodInvocation(this);
  }
}
```

E.4.29 Modifier.java

```
package translator.javaast;

import translator.lib.*;

public class Modifier implements JavaElement {
    public static final Modifier PUBLIC = new Modifier("public")
        ;
    public static final Modifier PRIVATE = new Modifier("private
        ");
    public static final Modifier STATIC = new Modifier("static")
        ;
    public static final Modifier ABSTRACT = new Modifier("
        abstract");

    private String text;

    private Modifier(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public String toString() {
        String result = "Modifier.";
        if(this == PUBLIC)
            result += "PUBLIC";
        else if(this == ABSTRACT)
            result += "ABSTRACT";
        else if(this == PRIVATE)
            result += "PRIVATE";
        else if(this == STATIC)
            result += "STATIC";
        else
            result += "UNKNOWN";
        return result;
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitModifier(this);
    }
}
```

E.4.30 NoPackageDeclaration.java

```
package translator.javaast;
```

```
import translator.lib.*;

public class NoPackageDeclaration extends
    OptionalPackageDeclaration
    implements JavaElement {

    public NoPackageDeclaration() { }

    public String toString() {
        return "NoPackageDeclaration()";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitNoPackageDeclaration(this);
    }
}
```

E.4.31 NullLiteral.java

```
package translator.javaast;

import translator.lib.*;

public class NullLiteral extends Expression implements
    JavaElement {
    public NullLiteral() { }

    public String toString() {
        return "NullLiteral()";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitNullLiteral(this);
    }
}
```

E.4.32 OptionalPackageDeclaration.java

```
package translator.javaast;

import translator.lib.*;

public abstract class OptionalPackageDeclaration implements
    JavaElement {

    public abstract String toString();

    public abstract void accept(JavaVisitor visitor);
```

```
}
```

E.4.33 PackageDeclaration.java

```
package translator.javaast;

import translator.lib.*;

public class PackageDeclaration extends
    OptionalPackageDeclaration
    implements JavaElement {
    private JavaName name;

    public PackageDeclaration(JavaName name) {
        this.name = name;
    }

    public JavaName getName() {
        return name;
    }

    public void setName(JavaName name) {
        this.name = name;
    }

    public String toString() {
        return "PackageDeclaration(" + name.toString() + ")";
    }


    public void accept(JavaVisitor visitor) {
        visitor.visitPackageDeclaration(this);
    }
}
```

E.4.34 ParentherizedExpression.java

```
package translator.javaast;

import translator.lib.*;

public class ParentherizedExpression extends Expression
    implements JavaElement {
    private Expression expression;

    public ParentherizedExpression(Expression expression) {
        this.expression = expression;
    }
```

```
    public Expression getExpression() {
        return expression;
    }

    public void setExpression(Expression expressionm) {
        this.expression = expression;
    }

    public String toString() {
        return "ParentherizedExpression(" + expression.toString
            () + ")";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitParentherizedExpression(this);
    }
}
```

E.4.35 PrefixExpression.java

```
package translator.javaast;

import translator.lib.*;

public class PrefixExpression extends Expression implements
    JavaElement {
    private PrefixOperator op;
    private Expression expression;

    public PrefixExpression(PrefixOperator op, Expression
        expression) {
        this.op = op;
        this.expression = expression;
    }

    public PrefixOperator getOp() {
        return op;
    }

    public void setOp(PrefixOperator op) {
        this.op = op;
    }

    public Expression getExpression() {
        return expression;
    }

    public void setRight(Expression expression) {
        this.expression = expression;
```

```
    }

    public String toString() {
        return "PrefixExpression(" + op.toString() + ", " +
            expression.toString() + ")";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitPrefixExpression(this);
    }
}
```

E.4.36 PrefixOperator.java

```
package translator.javaast;

import translator.lib.*;

public class PrefixOperator implements JavaElement {
    public static final PrefixOperator JAVA_PREFIX_OP_NOT =
        new PrefixOperator("!");

    private String text;

    private PrefixOperator(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public String toString() {
        String result = "PrefixOperator.";
        if(this == JAVA_PREFIX_OP_NOT)
            result += "JAVA_PREFIX_OP_NOT";
        else
            result += "UNKNOWN";
        return result;
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitPrefixOperator(this);
    }
}
```

E.4.37 PrimitiveType.java

```
package translator.javaast;
```

```
import translator.lib.*;

public class PrimitiveType extends JavaType implements
    JavaElement {
    public static final PrimitiveType JAVA_BOOLEAN = new
        PrimitiveType("boolean");
    public static final PrimitiveType JAVA_CHAR = new
        PrimitiveType("char");
    public static final PrimitiveType JAVA_DOUBLE = new
        PrimitiveType("double");
    public static final PrimitiveType JAVA_INT = new
        PrimitiveType("int");
    public static final PrimitiveType JAVA_VOID = new
        PrimitiveType("void");

    private String text;

    private PrimitiveType(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public String toString() {
        String result = "PrimitiveType.";
        if(this == JAVA_BOOLEAN)
            result += "JAVA_BOOLEAN";
        else if(this == JAVA_CHAR)
            result += "JAVA_CHAR";
        else if(this == JAVA_DOUBLE)
            result += "JAVA_DOUBLE";
        else if(this == JAVA_INT)
            result += "JAVA_INT";
        else if(this == JAVA_VOID)
            result += "JAVA_VOID";

        return result;
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitPrimitiveType(this);
    }
}
```

E.4.38 QualifiedName.java

```
package translator.javaast;

import translator.lib.*;

public class QualifiedName extends JavaName implements
    JavaElement {
    private JavaName left;
    private SimpleName right;

    public QualifiedName(JavaName left, SimpleName right) {
        this.left = left;
        this.right = right;
    }

    public JavaName getLeft() {
        return left;
    }

    public void setLeft(JavaName left) {
        this.left = left;
    }

    public SimpleName getRight() {
        return right;
    }

    public void setRight(SimpleName right) {
        this.right = right;
    }

    public String toString() {
        return "QualifiedName(" + left.toString() + ", " + right
            .toString() + ")";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitQualifiedName(this);
    }
}
```

E.4.39 ReferenceType.java

```
package translator.javaast;

import translator.lib.*;

public class ReferenceType
    extends JavaType
    implements JavaElement {
```

```
  private JavaName name;
  private ReferenceType optionalTypeArgument;

  public ReferenceType(JavaName name, ReferenceType
      optionalTypeArgument) {
    this.name = name;
    this.optionalTypeArgument = optionalTypeArgument;
  }

  public JavaName getName() {
    return name;
  }

  public void setName(JavaName name) {
    this.name = name;
  }

  public ReferenceType getOptionalTypeArgument() {
    return optionalTypeArgument;
  }

  public void setOptionalTypeArgument(ReferenceType
      optionalTypeArgument) {
    this.optionalTypeArgument = optionalTypeArgument;
  }

  public String toString() {
    return "ReferenceType(" + name.toString() + ", " +
        (optionalTypeArgument != null ? optionalTypeArgument.
            toString() :
         "noOptionalTypeArgument()") + ")";
  }

  public void accept(JavaVisitor visitor) {
    visitor.visitReferenceType(this);
  }
}
```

E.4.40 ReturnStatement.java

```
package translator.javaast;

import translator.lib.*;

public class ReturnStatement extends Statement implements
    JavaElement {
    private Expression expression;

    public ReturnStatement(Expression expression) {
```

```
        this.expression = expression;
    }

    public Expression getExpression() {
        return expression;
    }

    public void setExpression(Expression expression) {
        this.expression = expression;
    }

    public String toString() {
        return "ReturnStatement(" + expression + ")";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitReturnStatement(this);
    }
}
```

E.4.41 SimpleName.java

```
package translator.javaast;

import translator.lib.*;

public class SimpleName extends JavaName implements JavaElement
    {
    private String text;

    public SimpleName(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    public String toString() {
        return "SimpleName(" + text + ")";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitSimpleName(this);
    }
```

```
}
```

E.4.42 SingleVariableDeclaration.java

```
package translator.javaast;

import translator.lib.*;
import translator.rsllib.*;

public class SingleVariableDeclaration
    extends VariableDeclaration
    implements JavaElement {
  private RSLList < Modifier > modifierList;
  private JavaType type;
  private SimpleName name;
  private Expression initialization;

  public SingleVariableDeclaration(RSLList < Modifier >
      modifierList,
                                   JavaType type, SimpleName
                                       name,
                                   Expression initialization) {
    this.modifierList = modifierList;
    this.type = type;
    this.name = name;
    this.initialization = initialization;
  }

  public RSLList < Modifier > getModifierList() {
    return modifierList;
  }

  public void setModifierList(RSLList < Modifier > modifierList)
      {
    this.modifierList = modifierList;
  }

  public JavaType getType() {
    return type;
  }

  public void setType(JavaType type) {
    this.type = type;
  }

  public SimpleName getName() {
    return name;
  }
```

```
  public void setName(SimpleName name) {
    this.name = name;
  }

  public Expression getInitialization() {
    return initialization;
  }

  public void setInitialization(Expression initialization) {
    this.initialization = initialization;
  }

  public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("SingleVariableDeclaration(");
    for (Modifier m : modifierList.getList()) {
      result.append(m.toString());
      result.append(", ");
    }
    if (!modifierList.getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(type.toString());
    result.append(", ");
    result.append(name.toString());
    result.append(", ");
    if (initialization != null) {
      result.append(initialization.toString());
    }
    else {
      result.append("noOptionalInitialization()");
    }
    result.append(")");

    return result.toString();
  }

  public void accept(JavaVisitor visitor) {
    visitor.visitSingleVariableDeclaration(this);
  }
}
```

E.4.43 Statement.java

```
package translator.javaast;

import translator.lib.*;

public abstract class Statement implements JavaElement {
```

```
    public abstract String toString();

    public abstract void accept(JavaVisitor visitor);
}
```

E.4.44 StringLiteral.java

```
package translator.javaast;

import translator.lib.*;

public class StringLiteral extends Expression implements
    JavaElement {
    private String text;

    public StringLiteral(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    public String toString() {
        return "StringLiteral(" + text + ")";
    }

    public void accept(JavaVisitor visitor) {
        visitor.visitStringLiteral(this);
    }
}
```

E.4.45 ThisExpression.java

```
package translator.javaast;

import translator.lib.*;

public class ThisExpression
    extends Expression
    implements JavaElement {
  private SimpleName simpleName;

  public ThisExpression(SimpleName simpleName) {
```

```
    this.simpleName = simpleName;
  }

  public SimpleName getSimpleName() {
    return simpleName;
  }

  public void setSimpleName(SimpleName simpleName) {
    this.simpleName = simpleName;
  }

  public String toString() {
    return "ThisExpression(" +
        (simpleName != null ? simpleName.toString() : "
            noSimpleName()") + ")";
  }

  public void accept(JavaVisitor visitor) {
    visitor.visitThisExpression(this);
  }
}
```

E.4.46 TypeDeclaration.java

```
package translator.javaast;

import translator.lib.*;
import translator.rsllib.*;

public abstract class TypeDeclaration implements JavaElement {

    public abstract  String toString();

    public abstract void accept(JavaVisitor visitor);
}
```

E.4.47 VariableDeclaration.java

```
package translator.javaast;

import translator.lib.*;

public abstract class VariableDeclaration  implements
    JavaElement {

    public abstract void accept(JavaVisitor visitor);

}
```

E.4.48 VariableDeclarationExpression.java

```
package translator.javaast;

import translator.lib.*;
import translator.rsllib.*;

public class VariableDeclarationExpression
    extends Expression
    implements JavaElement {
  private RSLList < Modifier > modifierList;
  private JavaType type;
  private SimpleName name;
  private Expression optionalInitialisation;

  public VariableDeclarationExpression(RSLList < Modifier >
      modifierList,
                                       JavaType type, SimpleName
                                           name,
                                       Expression
                                           optionalInitialisation
                                           ) {
    this.modifierList = modifierList;
    this.type = type;
    this.name = name;
    this.optionalInitialisation = optionalInitialisation;
  }

  public RSLList < Modifier > getModifierList() {
    return modifierList;
  }

  public void setModifierList(RSLList < Modifier > modifierList)
      {
    this.modifierList = modifierList;
  }

  public JavaType getType() {
    return type;
  }

  public void setType(JavaType type) {
    this.type = type;
  }

  public SimpleName getName() {
    return name;
  }
```

```
  public void setName(SimpleName name) {
    this.name = name;
  }

  public Expression getOptionalInitialisation() {
    return optionalInitialisation;
  }

  public void setOptionalInitialisation(Expression
      optinalInitialisation) {
    this.optionalInitialisation = optionalInitialisation;
  }

  public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("VariableDeclarationExpression(");
    result.append("ModifierList(");
    for (Modifier m : modifierList.getList()) {
      result.append(m.toString());
      result.append(", ");
    }
    if (!modifierList.getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append("),");
    result.append(type.toString());
    result.append(", ");
    result.append(name.toString());
    result.append(", ");
    result.append(optionalInitialisation.toString());
    result.append(")");

    return result.toString();
  }

  public void accept(JavaVisitor visitor) {
    visitor.visitVariableDeclarationExpression(this);
  }
}
```

E.4.49 VariableDeclarationFragment.java

```
package translator.javaast;

import translator.lib.*;

public class VariableDeclarationFragment
    extends VariableDeclaration
    implements JavaElement {
```

```
  private SimpleName name;
  private Expression initialization;

  public VariableDeclarationFragment(SimpleName name, Expression
      initialization) {
    this.name = name;
    this.initialization = initialization;
  }

  public SimpleName getName() {
    return name;
  }

  public void setName(SimpleName name) {
    this.name = name;
  }

  public Expression getInitialization() {
    return initialization;
  }

  public void setInitialization(Expression initialization) {
    this.initialization = initialization;
  }

  public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("VariableDeclarationFragment(");
    result.append(name.toString());
    if (initialization != null) {
      result.append(", ");
      result.append(initialization.toString());
    }
    result.append(")");

    return result.toString();
  }

  public void accept(JavaVisitor visitor) {
    visitor.visitVariableDeclarationFragment(this);
  }
}
```

E.4.50 VariableDeclarationStatement.java

```
package translator.javaast;

import translator.lib.*;
import translator.rsllib.*;
```

```
5
public class VariableDeclarationStatement
    extends Statement
    implements JavaElement {
  private RSLList < Modifier > modifierList;
10  private JavaType type;
  private VariableDeclarationFragment fragment;

  public VariableDeclarationStatement(RSLList < Modifier >
      modifierList,
                                      JavaType type,
15                                    VariableDeclarationFragment
                                          fragment) {
    this.modifierList = modifierList;
    this.type = type;
    this.fragment = fragment;
  }
20
  public RSLList < Modifier > getModifierList() {
    return modifierList;
  }

25  public void setModifierList(RSLList < Modifier > modifierList)
      {
    this.modifierList = modifierList;
  }

  public JavaType getType() {
30    return type;
  }

  public void setType(JavaType type) {
    this.type = type;
35  }

  public VariableDeclarationFragment getFragment() {
    return fragment;
  }
40
  public void setFragment(VariableDeclarationFragment fragment)
      {
    this.fragment = fragment;
  }

45  public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("VariableDeclarationStatement(");
    result.append("ModifierList(");
    for (Modifier m : modifierList.getList()) {
```

```
50        result.append(m.toString());
          result.append(", ");
        }
        if (!modifierList.getList().isEmpty()) {
          result.delete(result.length() - 2, result.length());
55      }
        result.append("),");
        result.append(type.toString());
        result.append(", ");
        result.append(fragment.toString());
60      result.append(")");

        return result.toString();
      }

65    public void accept(JavaVisitor visitor) {
        visitor.visitVariableDeclarationStatement(this);
      }
    }
```

E.5 translator.lib

E.5.1 AST.java

```
package translator.lib;

public abstract class AST implements Element {


5
}
```

E.5.2 Element.java

```
package translator.lib;

public interface Element {
    public void accept(Visitor visitor);
5   public Element getParent();
}
```

E.5.3 JavaElement.java

```
package translator.lib;

public interface JavaElement {
    public void accept(JavaVisitor visitor);
5 }
```

E.5.4 JavaVisitor.java

```
package translator.lib;

import translator.javaast.*;

public abstract class JavaVisitor {

  public void visitJavaAst(JavaAst javaAst) {
    for (String s : javaAst.getCompilationUnitMap().getMap().
        keySet()) {
      javaAst.getCompilationUnitMap().getMap().get(s).accept(
          this);
    }
  }

  public void visitCompilationUnit(CompilationUnit
      compilationUnit) {
    compilationUnit.getOptionalPackageDeclaration().accept(this)
        ;
    for (ImportDeclaration id :
        compilationUnit.getImportDeclarationList().getList()) {
      id.accept(this);
    }
    for (TypeDeclaration td : compilationUnit.
        getTypeDeclarationList().getList()) {
      td.accept(this);
    }
  }

  public void visitPackageDeclaration(PackageDeclaration
      packageDeclaration) {
    packageDeclaration.getName().accept(this);
  }

  public void visitNoPackageDeclaration(NoPackageDeclaration
                                        noPackageDeclaration) {}

  public void visitImportDeclaration(ImportDeclaration
      importDeclaration) {
    importDeclaration.getName().accept(this);
  }

  public void visitClassDeclaration(ClassDeclaration
      classDeclaration) {
    for (Modifier m : classDeclaration.getModifierList().getList
        ()) {
      m.accept(this);
    }
```

```
    classDeclaration.getSimpleName().accept(this);
    if (classDeclaration.getOptionalExtend() != null) {
      classDeclaration.getOptionalExtend().accept(this);
    }
    for (SimpleName sn : classDeclaration.getImplementList().
        getList()) {
      sn.accept(this);
    }
    for (ConstructorDeclaration cd :
        classDeclaration.getConstructorDeclarationList().
            getList()) {
      cd.accept(this);
    }
    for (MethodDeclaration md :
        classDeclaration.getMethodDeclarationList().getList())
            {
      md.accept(this);
    }
    for (FieldDeclaration fd :
        classDeclaration.getFieldDeclarationList().getList()) {
      fd.accept(this);
    }
    for (ClassDeclaration cd :
        classDeclaration.getClassDeclarationList().getList()) {
      cd.accept(this);
    }
  }

  public void visitMethodDeclaration(MethodDeclaration
      methodDeclaration) {
    for (Modifier m : methodDeclaration.getModifierList().
        getList()) {
      m.accept(this);
    }
    methodDeclaration.getReturnType().accept(this);
    methodDeclaration.getSimpleName().accept(this);
    for (SingleVariableDeclaration v :
        methodDeclaration.getArgumentList().getList()) {
      v.accept(this);
    }
    if (methodDeclaration.getBlock() != null) {
      methodDeclaration.getBlock().accept(this);
    }
  }

  public void visitConstructorDeclaration(ConstructorDeclaration
                                          constructorDeclaration
                                              ) {
    for (Modifier m : constructorDeclaration.getModifierList().
```

```
        getList()) {
      m.accept(this);
    }
    constructorDeclaration.getSimpleName().accept(this);
    for (SingleVariableDeclaration v :
        constructorDeclaration.getArgumentList().getList()) {
      v.accept(this);
    }
    if (constructorDeclaration.getBlock() != null) {
      constructorDeclaration.getBlock().accept(this);
    }
  }

  public void visitFieldDeclaration(FieldDeclaration
      fieldDeclaration) {
    for (Modifier m : fieldDeclaration.getModifierList().getList
        ()) {
      m.accept(this);
    }
    fieldDeclaration.getType().accept(this);
    fieldDeclaration.getVariableDeclarationFragment().accept(
        this);
  }

  public void visitSingleVariableDeclaration(
      SingleVariableDeclaration
                                          singleVariableDeclaration
                                              ) {
    for (Modifier m : singleVariableDeclaration.getModifierList
        ().getList()) {
      m.accept(this);
    }
    singleVariableDeclaration.getType().accept(this);
    singleVariableDeclaration.getName().accept(this);
    if (singleVariableDeclaration.getInitialization() != null) {
      singleVariableDeclaration.getInitialization().accept(this)
          ;
    }
  }

  public void visitVariableDeclarationFragment(
      VariableDeclarationFragment
                                            variableDeclarationFragment
                                                ) {
    variableDeclarationFragment.getName().accept(this);
    if (variableDeclarationFragment.getInitialization() != null)
        {
      variableDeclarationFragment.getInitialization().accept(
          this);
```

```
    }
  }

  public void visitBlock(Block block) {
    for (Statement s : block.getStatementList().getList()) {
      s.accept(this);
    }
  }

  //Type
  public void visitPrimitiveType(PrimitiveType primitiveType) {}

  public void visitReferenceType(ReferenceType referenceType) {
    referenceType.getName().accept(this);
    if (referenceType.getOptionalTypeArgument() != null) {
      referenceType.getOptionalTypeArgument().accept(this);
    }
  }

  public void visitArrayType(ArrayType arrayType) {
    arrayType.getType().accept(this);
  }

  //Statements
  public void visitExpressionStatement(ExpressionStatement
      expressionStatement) {
    expressionStatement.getExpression().accept(this);
  }

  public void visitIfStatement(IfStatement ifStatement) {
    ifStatement.getCondition().accept(this);
    ifStatement.getIfBlock().accept(this);
    if (ifStatement.getOptionalElseBlock() != null) {
      ifStatement.getOptionalElseBlock().accept(this);
    }
  }

  public void visitReturnStatement(ReturnStatement
      returnStatement) {
    returnStatement.getExpression().accept(this);
  }

  public void visitVariableDeclarationStatement(
      VariableDeclarationStatement
                                                variableDeclarationStatement
                                                    ) {}

  //Expressions
  public void visitMethodInvocation(MethodInvocation
```

```
        methodInvocation) {
      if (methodInvocation.getOptionalExpression() != null) {
165     methodInvocation.getOptionalExpression().accept(this);
      }
      methodInvocation.getName().accept(this);
      for (Expression e : methodInvocation.getArgumentList().
          getList()) {
        e.accept(this);
170   }
    }

    public void visitInfixExpression(InfixExpression
        infixExpression) {
      infixExpression.getLeft().accept(this);
175   infixExpression.getOp().accept(this);
      infixExpression.getRight().accept(this);
    }

    public void visitPrefixExpression(PrefixExpression
        prefixExpression) {
180   prefixExpression.getOp().accept(this);
      prefixExpression.getExpression().accept(this);
    }

    public void visitAssignmentExpression(AssignmentExpression
185                                       assignmentExpression) {
      assignmentExpression.getLeftHandSide().accept(this);
      assignmentExpression.getOp().accept(this);
      assignmentExpression.getRightHandSide().accept(this);
    }
190
    public void visitClassInstanceCreation(ClassInstanceCreation
                                           classInstanceCreation)
                                               {
      if (classInstanceCreation.getExpression() != null) {
        classInstanceCreation.getExpression().accept(this);
195   }
      classInstanceCreation.getType().accept(this);
      for (Expression e : classInstanceCreation.getArgumentList().
          getList()) {
        e.accept(this);

200   }
    }

    public void visitInstanceOfExpression(InstanceOfExpression
                                          instanceOfExpression) {
205   instanceOfExpression.getExpression().accept(this);
      instanceOfExpression.getType().accept(this);
```

```
  }

  public void visitCastExpression(CastExpression castExpression)
      {
    castExpression.getType().accept(this);
    castExpression.getExpression().accept(this);
  }

  public void visitParentherizedExpression(
      ParentherizedExpression
                                             parentherizedExpression
                                                ) {
    parentherizedExpression.getExpression().accept(this);
  }

  public void visitFieldAccessExpression(FieldAccessExpression
                                           fieldAccessExpression)
                                              {
    if (fieldAccessExpression.getExpression() != null) {
      fieldAccessExpression.getExpression().accept(this);
    }
    fieldAccessExpression.getSimpleName().accept(this);
  }

  public void visitThisExpression(ThisExpression thisExpression)
      {
    if (thisExpression.getSimpleName() != null) {
      thisExpression.getSimpleName().accept(this);
    }
  }

  public void visitArrayCreation(ArrayCreation arrayCreation) {
    arrayCreation.getType().accept(this);
    if (arrayCreation.getExpression() != null) {
      arrayCreation.getExpression().accept(this);
    }
    if (arrayCreation.getExpressionList() != null) {
      for (Expression expression : arrayCreation.
          getExpressionList().getList()) {
        expression.accept(this);
      }
    }
  }

  public void visitVariableDeclarationExpression(
      VariableDeclarationExpression
                                                   variableDeclarationExpression
                                                      ) {}
```

```
    public void visitBooleanLiteral(BooleanLiteral booleanLiteral)
        {}

    public void visitIntegerLiteral(IntegerLiteral integerLiteral)
        {}

    public void visitDoubleLiteral(DoubleLiteral doubleLiteral) {}

    public void visitCharLiteral(CharLiteral charLiteral) {}

    public void visitStringLiteral(StringLiteral stringLiteral) {}

    public void visitNullLiteral(NullLiteral nullLiteral) {}

    //Other
    public void visitQualifiedName(QualifiedName qualifiedName) {
      qualifiedName.getLeft().accept(this);
      qualifiedName.getRight().accept(this);
    }

    public void visitModifier(Modifier modifier) {}

    public void visitSimpleName(SimpleName simpleName) {}

    public void visitInfixOperator(InfixOperator infixOperator) {}

    public void visitPrefixOperator(PrefixOperator prefixOperator)
        {}

    public void visitAssignmentOperator(AssignmentOperator
       assignmentOperator) {}
}
```

E.5.5 ParentVisitor.java

```
package translator.lib;

import translator.rslast.*;

public class ParentVisitor
    extends Visitor {
  public void visitRSLAst(RSLAst rslast) {
    rslast.getLibModule().accept(this);
    rslast.getLibModule().setParent(rslast);
    rslast.setParent(null);
  }

  public void visitLibModule(LibModule module) {
    module.getSchemeDef().accept(this);
```

```
15      module.getSchemeDef().setParent(module);
    }

    public void visitSchemeDef(SchemeDef schemeDef) {
      schemeDef.getClassExpr().accept(this);
20      schemeDef.getClassExpr().setParent(schemeDef);
    }

    public void visitBasicClassExpr(BasicClassExpr basicClassExpr)
        {
      for (Decl decl : basicClassExpr.getDeclList().getList()) {
25      decl.accept(this);
        decl.setParent(basicClassExpr);
      }
    }

30  /*Type Declarations*/
    public void visitTypeDecl(TypeDecl typeDecl) {
      for (TypeDef typeDef : typeDecl.getTypeDefList().getList())
          {
        typeDef.accept(this);
      }
35  }

    public void visitSortDef(SortDef sortDef) {
      sortDef.getId().accept(this);
    }
40
    public void visitVariantDef(VariantDef variantDef) {
      variantDef.getId().accept(this);
      for (Variant variant : variantDef.getVariantList().getList()
          ) {
        variant.accept(this);
45      variant.setParent(variantDef);
      }
    }

    public void visitShortRecordDef(ShortRecordDef shortRecordDef)
        {
50    shortRecordDef.getId().accept(this);
      for (ComponentKind componentKind :
          shortRecordDef.getComponentKindString().getList()) {
        componentKind.accept(this);
        componentKind.setParent(shortRecordDef);
55    }
    }

    public void visitConstructor(Constructor constructor) {
      constructor.getId().accept(this);
```

```
60    }

      public void visitDestructor(Destructor destructor) {
        destructor.getId().accept(this);
      }
65
      public void visitRecordVariant(RecordVariant recordVariant) {
        recordVariant.getConstructor().accept(this);
        recordVariant.getConstructor().setParent(recordVariant);
        for (ComponentKind componentKind :
70           recordVariant.getComponentKindList().getList()) {
          componentKind.accept(this);
          componentKind.setParent(recordVariant);
        }
      }
75
      public void visitComponentKind(ComponentKind componentKind) {
        componentKind.getTypeExpr().accept(this);
        if (componentKind.getOptionalDestructor() != null) {
          componentKind.getOptionalDestructor().accept(this);
80      }
        if (componentKind.getOptionalReconstructor() != null) {
          componentKind.getOptionalReconstructor().accept(this);
        }
      }
85
      /*Value Declarations*/
      public void visitValueDecl(ValueDecl valueDecl) {
        for (ValueDef valueDef : valueDecl.getValueDefList().getList
            ()) {
          valueDef.accept(this);
90        valueDef.setParent(valueDecl);
        }
      }

      public void visitExplicitFunctionDef(ExplicitFunctionDef
          explicitFunctionDef) {
95      explicitFunctionDef.getSingleTyping().accept(this);
        explicitFunctionDef.getFormalFunctionApplication().accept(
            this);
        explicitFunctionDef.getValueExpr().accept(this);
        explicitFunctionDef.getOptionalPrecondition().accept(this);

100     explicitFunctionDef.getSingleTyping().setParent(
            explicitFunctionDef);
        explicitFunctionDef.getFormalFunctionApplication().setParent
            (
             explicitFunctionDef);
        explicitFunctionDef.getValueExpr().setParent(
```

```
        explicitFunctionDef);
    explicitFunctionDef.getOptionalPrecondition().setParent(
        explicitFunctionDef);
  }

  public void visitSingleTyping(SingleTyping singleTyping) {
    singleTyping.getBinding().accept(this);
    singleTyping.getTypeExpr().accept(this);

    singleTyping.getBinding().setParent(singleTyping);
    singleTyping.getTypeExpr().setParent(singleTyping);
  }

  public void visitIdApplication(IdApplication idApplication) {
    idApplication.getId().accept(this);
    for (FormalFunctionParameter parameter :
         idApplication.getFormalFunctionParameters().getList())
             {
      parameter.accept(this);
    }

    idApplication.getId().setParent(idApplication);
    for (FormalFunctionParameter parameter :
         idApplication.getFormalFunctionParameters().getList())
             {
      parameter.setParent(idApplication);
    }
  }

  public void visitFormalFunctionParameter(
     FormalFunctionParameter
                                           formalFunctionParameter
                                              ) {
    for (Binding binding : formalFunctionParameter.
        getBindingList().getList()) {
      binding.accept(this);
    }

    for (Binding binding : formalFunctionParameter.
        getBindingList().getList()) {
      binding.setParent(formalFunctionParameter);
    }
  }

  public void visitPrecondition(Precondition precondition) {}

  public void visitNoPrecondition(NoPrecondition noPrecondition)
      {}
```

```
    //Type Expressions
145 public void visitFunctionTypeExpr(FunctionTypeExpr
      functionTypeExpr) {
    functionTypeExpr.getTypeExpr().accept(this);
    functionTypeExpr.getFunctionArrow().accept(this);
    functionTypeExpr.getFunctionResultDescription().accept(this)
        ;

150 functionTypeExpr.getTypeExpr().setParent(functionTypeExpr);
    functionTypeExpr.getFunctionArrow().setParent(
        functionTypeExpr);
    functionTypeExpr.getFunctionResultDescription().setParent(
        functionTypeExpr);
  }

155 public void visitFunctionArrow(FunctionArrow functionArrow) {}

  public void visitFunctionResultDescription(
      FunctionResultDescription
                                              functionResultDescription
                                                  ) {
    functionResultDescription.getOptionalAccessDescription().
        accept(this);
160 functionResultDescription.getTypeExpr().accept(this);

    functionResultDescription.getOptionalAccessDescription().
        setParent(
         functionResultDescription);
    functionResultDescription.getTypeExpr().setParent(
        functionResultDescription);
165 }

  public void visitNoAccessDescription(NoAccessDescription
      noAccessDescription) {}

  public void visitFiniteListTypeExpr(FiniteListTypeExpr
      finiteListExpr) {
170 finiteListExpr.getTypeExpr().accept(this);

    finiteListExpr.getTypeExpr().setParent(finiteListExpr);
  }

175 public void visitProductTypeExpr(ProductTypeExpr
      productTypeExpr) {
    for (TypeExpr te : productTypeExpr.getTypeExprList().getList
        ()) {
      te.accept(this);
    }
```

```
180     for (TypeExpr te : productTypeExpr.getTypeExprList().getList
            ()) {
          te.setParent(productTypeExpr);
        }
      }

185   public void visitTypeLiteral(TypeLiteral typeLiteral) {}

      public void visitId(Id id) {}

      //Value Expressions
190   public void visitApplicationExpr(ApplicationExpr
          applicationExpr) {
        applicationExpr.getValueExpr().accept(this);
        applicationExpr.getValueExpr().setParent(applicationExpr);
        for (ValueExpr ve : applicationExpr.getOptionalValueExprList
            ().getList()) {
          ve.accept(this);
195       ve.setParent(applicationExpr);
        }
      }

      public void visitEnumeratedListExpr(EnumeratedListExpr
          enumeratedListExpr) {
200     for (ValueExpr ve : enumeratedListExpr.getValueExprList().
            getList()) {
          ve.accept(this);
        }

        for (ValueExpr ve : enumeratedListExpr.getValueExprList().
            getList()) {
205       ve.setParent(enumeratedListExpr);
        }
      }

      public void visitIfExpr(IfExpr ifExpr) {
210     ifExpr.getCondition().accept(this);
        ifExpr.getValueExpr().accept(this);
        for (ElsifBranch elsif : ifExpr.getElsifBranchList().getList
            ()) {
          elsif.accept(this);
        }
215     ifExpr.getElseBranch().accept(this);
        ifExpr.getCondition().setParent(ifExpr);
        ifExpr.getValueExpr().setParent(ifExpr);
        for (ElsifBranch elsif : ifExpr.getElsifBranchList().getList
            ()) {
          elsif.setParent(ifExpr);
220     }
```

```
    ifExpr.getElseBranch().setParent(ifExpr);
  }

  public void visitElsifBranch(ElsifBranch elsifBranch) {
    elsifBranch.getCondition().accept(this);
    elsifBranch.getValueExpr().accept(this);

    elsifBranch.getCondition().setParent(elsifBranch);
    elsifBranch.getValueExpr().setParent(elsifBranch);
  }

  public void visitElseBranch(ElseBranch elseBranch) {
    elseBranch.getValueExpr().accept(this);
    elseBranch.getValueExpr().setParent(elseBranch);
  }

  public void visitCaseExpr(CaseExpr caseExpr) {
    caseExpr.getCondition().accept(this);
    for (CaseBranch cb : caseExpr.getCaseBranchList().getList())
        {
      cb.accept(this);
      cb.setParent(caseExpr);
    }
  }

  public void visitCaseBranch(CaseBranch caseBranch) {
    caseBranch.getPattern().accept(this);
    caseBranch.getValueExpr().accept(this);
    caseBranch.getPattern().setParent(caseBranch);
    caseBranch.getValueExpr().setParent(caseBranch);
  }

  public void visitNamePattern(NamePattern namePattern) {
    namePattern.getId().accept(this);
  }

  public void visitRecordPattern(RecordPattern recordPattern) {
    recordPattern.getValueOrVariableName().accept(this);
    for (Pattern p : recordPattern.getInnerPatternList().getList
        ()) {
      p.accept(this);
    }
  }

  public void visitValueLiteralPattern(ValueLiteralPattern
      valueLiteralPattern) {
    valueLiteralPattern.getValueLiteral().accept(this);
  }
```

```
    public void visitWildcardPattern(WildcardPattern
        wildcardPattern) {}

    public void visitValueInfixExpr(ValueInfixExpr valueInfixExpr)
        {
      valueInfixExpr.getLeft().accept(this);
      valueInfixExpr.getOp().accept(this);
      valueInfixExpr.getRight().accept(this);

      valueInfixExpr.getLeft().setParent(valueInfixExpr);
      valueInfixExpr.getOp().setParent(valueInfixExpr);
      valueInfixExpr.getRight().setParent(valueInfixExpr);
    }

    public void visitValuePrefixExpr(ValuePrefixExpr
        valuePrefixExpr) {
      valuePrefixExpr.getOp().accept(this);
      valuePrefixExpr.getValueExpr().accept(this);

      valuePrefixExpr.getOp().setParent(valuePrefixExpr);
      valuePrefixExpr.getValueExpr().setParent(valuePrefixExpr);
    }

    public void visitRSLInfixOp(RSLInfixOp rslInfixOp) {}

    public void visitRSLPrefixOp(RSLPrefixOp rslPrefixOp) {}

    public void visitValueOrVariableName(ValueOrVariableName
        valueOrVariableName) {
      valueOrVariableName.getId().accept(this);

      valueOrVariableName.getId().setParent(valueOrVariableName);
    }

    public void visitValueLiteralInteger(ValueLiteralInteger
        valueLiteralInteger) {}

    public void visitValueLiteralReal(ValueLiteralReal
        valueLiteralReal) {}
}
```

E.5.6 RSLElement.java

```
package translator.lib;

import translator.rslast.*;

public abstract class RSLElement
    implements Element {
```

```
  private RSLElement parent;
  private TypeEvaluator typeEvaluator;

  public RSLElement getParent() {
    return parent;
  }

  public void setParent(RSLElement parent) {
    this.parent = parent;
  }

  public TypeEvaluator getTypeEvaluatorForTypeEvaluation() {
    return typeEvaluator;
  }

  public void setTypeEvaluatorForTypeEvaluation(TypeEvaluator
      typeEvaluator) {
    this.typeEvaluator = typeEvaluator;
  }

  public abstract void accept(Visitor visitor);
}
```

E.5.7 StringJavaVisitor.java

```
package translator.lib;

import translator.javaast.*;

import java.io.*;
import java.text.*;
import java.util.*;

public class StringJavaVisitor
    extends JavaVisitor {
  private StringBuffer result;
  private boolean writeFiles;
  private String directory;
  private String extraMethodString;

  public StringJavaVisitor() {
    this.result = new StringBuffer();
    this.writeFiles = false;
  }

  public StringJavaVisitor(boolean writeFiles, String directory,
                           String extraMethodString) {
    this.result = new StringBuffer();
    this.writeFiles = writeFiles;
```

```
25      this.directory = directory;
        this.extraMethodString = extraMethodString;
    }

    public void visitJavaAst(JavaAst javaAst) {
30      //result.append("JAVAAST\n");
        super.visitJavaAst(javaAst);
    }

    public void visitCompilationUnit(CompilationUnit
        compilationUnit) {
35      if (writeFiles) {
          this.result = new StringBuffer();
        }
        super.visitCompilationUnit(compilationUnit);
        if (writeFiles) {
40        for (TypeDeclaration td :
              compilationUnit.getTypeDeclarationList().getList()) {
            if (td instanceof ClassDeclaration) {
              ClassDeclaration cd = (ClassDeclaration) td;
              for (Modifier m : cd.getModifierList().getList()) {
45              if (m == Modifier.PUBLIC) {
                  try {
                    FileWriter fw = new FileWriter(directory + "/" +
                                                   cd.getSimpleName
                                                       ().getText() +
                                                   ".java", false);
50                  fw.write("//" +
                             DateFormat.getDateTimeInstance(
                                 DateFormat.LONG,
                         DateFormat.LONG, Locale.UK).format(new Date
                             ()) + "\n");
                    fw.write(this.result.toString());
                    fw.flush();
55                  fw.close();
                  }
                  catch (IOException ioe) {
                    ioe.printStackTrace();
                  }
60              }
              }
            }
          }
        }
65  }

    public void visitPackageDeclaration(PackageDeclaration
        packageDeclaration) {
      result.append("package ");
```

```
    super.visitPackageDeclaration(packageDeclaration);
    result.append(";\n");
  }

  public void visitNoPackageDeclaration(NoPackageDeclaration
                                        noPackageDeclaration) {
    //result.append("NO PACKAGE DECLARATION\n");
  }

  public void visitImportDeclaration(ImportDeclaration
      importDeclaration) {
    result.append("import ");
    super.visitImportDeclaration(importDeclaration);
    result.append(";\n");
  }

  public void visitClassDeclaration(ClassDeclaration
      classDeclaration) {
    for (Modifier m : classDeclaration.getModifierList().getList
        ()) {
      m.accept(this);
      result.append(" ");
    }
    result.append("class ");
    classDeclaration.getSimpleName().accept(this);
    if (classDeclaration.getOptionalExtend() != null) {
      result.append(" extends ");
      classDeclaration.getOptionalExtend().accept(this);
    }
    if (!classDeclaration.getImplementList().getList().isEmpty()
        ) {
      result.append(" implements ");
    }
    for (SimpleName sn : classDeclaration.getImplementList().
        getList()) {
      sn.accept(this);
      result.append(", ");
    }
    if (!classDeclaration.getImplementList().getList().isEmpty()
        ) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(" {\n");
    for (FieldDeclaration fd :
        classDeclaration.getFieldDeclarationList().getList()) {
      fd.accept(this);
    }
    for (ConstructorDeclaration cd :
        classDeclaration.getConstructorDeclarationList().
```

```
            getList()) {
        cd.accept(this);
      }
      for (MethodDeclaration md :
115         classDeclaration.getMethodDeclarationList().getList())
              {
        md.accept(this);
      }
      for (ClassDeclaration cd :
          classDeclaration.getClassDeclarationList().getList()) {
120     cd.accept(this);
      }
      if (extraMethodString != null) {
        result.append(extraMethodString);
      }
125   result.append("\n}\n");
    }

    public void visitFieldDeclaration(FieldDeclaration
        fieldDeclaration) {
      for (Modifier m : fieldDeclaration.getModifierList().getList
          ()) {
130     m.accept(this);
        result.append(" ");
      }
      fieldDeclaration.getType().accept(this);
      result.append(" ");
135   fieldDeclaration.getVariableDeclarationFragment().accept(
          this);
      result.append(";\n");
    }

    public void visitMethodDeclaration(MethodDeclaration
        methodDeclaration) {
140   for (Modifier m : methodDeclaration.getModifierList().
          getList()) {
        m.accept(this);
        result.append(" ");
      }
      methodDeclaration.getReturnType().accept(this);
145   result.append(" ");
      methodDeclaration.getSimpleName().accept(this);
      result.append("(");
      boolean hasElements = false;
      for (SingleVariableDeclaration v :
150       methodDeclaration.getArgumentList().getList()) {
        v.accept(this);
        result.append(", ");
        hasElements = true;
```

```
    }
    if (hasElements) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")");
    if (methodDeclaration.getBlock() != null) {
      methodDeclaration.getBlock().accept(this);
    }
    else {
      result.append(";");
    }
    result.append("\n\n");
  }

  public void visitConstructorDeclaration(ConstructorDeclaration
                                          constructorDeclaration
                                              ) {
    for (Modifier m : constructorDeclaration.getModifierList().
        getList()) {
      m.accept(this);
      result.append(" ");
    }
    constructorDeclaration.getSimpleName().accept(this);
    result.append("(");
    boolean hasElements = false;
    for (SingleVariableDeclaration v :
         constructorDeclaration.getArgumentList().getList()) {
      v.accept(this);
      result.append(", ");
      hasElements = true;
    }
    if (hasElements) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")");
    constructorDeclaration.getBlock().accept(this);
    result.append("\n");
  }

  public void visitSingleVariableDeclaration(
      SingleVariableDeclaration
                                             singleVariableDeclaration
                                                 ) {
    for (Modifier m : singleVariableDeclaration.getModifierList
        ().getList()) {
      m.accept(this);
      result.append(" ");
    }
    singleVariableDeclaration.getType().accept(this);
```

```
      result.append(" ");
      singleVariableDeclaration.getName().accept(this);
      if (singleVariableDeclaration.getInitialization() != null) {
        result.append(" = ");
        singleVariableDeclaration.getInitialization().accept(this)
            ;
      }
    }

    public void visitVariableDeclarationFragment(
        VariableDeclarationFragment
                                                  variableDeclarationFragment
                                                      ) {
      variableDeclarationFragment.getName().accept(this);
      if (variableDeclarationFragment.getInitialization() != null)
          {
        result.append(" = ");
        variableDeclarationFragment.getInitialization().accept(
            this);
      }
    }

    public void visitBlock(Block block) {
      result.append("{\n");
      super.visitBlock(block);
      result.append("\n}");
    }

    //Type
    public void visitPrimitiveType(PrimitiveType primitiveType) {
      result.append(primitiveType.getText());
    }

    public void visitReferenceType(ReferenceType referenceType) {
      referenceType.getName().accept(this);
      if (referenceType.getOptionalTypeArgument() != null) {
        result.append("<");
        referenceType.getOptionalTypeArgument().accept(this);
        result.append(">");
      }
    }

    public void visitArrayType(ArrayType arrayType) {
      arrayType.getType().accept(this);
      result.append("[]");
    }

    //Statements
    public void visitExpressionStatement(ExpressionStatement
```

```
      expressionStatement) {
    expressionStatement.getExpression().accept(this);
    result.append(";\n");
  }

  public void visitIfStatement(IfStatement ifStatement) {
    result.append("if(");
    ifStatement.getCondition().accept(this);
    result.append(")\n");
    ifStatement.getIfBlock().accept(this);
    result.append("\n");
    if (ifStatement.getOptionalElseBlock() != null) {
      result.append("else");
      ifStatement.getOptionalElseBlock().accept(this);
      result.append("\n");
    }
  }

  public void visitReturnStatement(ReturnStatement
      returnStatement) {
    result.append("return ");
    returnStatement.getExpression().accept(this);
    result.append(";");
  }

  public void visitVariableDeclarationStatement(
      VariableDeclarationStatement
                                                variableDeclarationStatement
                                                  ) {
    for (Modifier m : variableDeclarationStatement.
        getModifierList().getList()) {
      m.accept(this);
      result.append(" ");
    }
    variableDeclarationStatement.getType().accept(this);
    result.append(" ");
    variableDeclarationStatement.getFragment().accept(this);
    result.append(";\n");
  }

  //Expressions
  public void visitMethodInvocation(MethodInvocation
      methodInvocation) {
    if (methodInvocation.getOptionalExpression() != null) {
      methodInvocation.getOptionalExpression().accept(this);
      result.append(".");
    }
    methodInvocation.getName().accept(this);
    result.append("(");
```

```
285     if (methodInvocation.getArgumentList() != null) {
          boolean hasElements = false;
          for (Expression e : methodInvocation.getArgumentList().
              getList()) {
            e.accept(this);
            result.append(", ");
            hasElements = true;
          }
          if (hasElements) {
            result.delete(result.length() - 2, result.length());
          }
        }
        result.append(")");
      }

      public void visitInstanceOfExpression(InstanceOfExpression
                                              instanceOfExpression) {
        instanceOfExpression.getExpression().accept(this);
        result.append(" instanceof ");
        instanceOfExpression.getType().accept(this);
      }

      public void visitInfixExpression(InfixExpression
          infixExpression) {
        infixExpression.getLeft().accept(this);
        result.append(" ");
        infixExpression.getOp().accept(this);
        result.append(" ");
        infixExpression.getRight().accept(this);
      }

      public void visitPrefixExpression(PrefixExpression
          prefixExpression) {
        prefixExpression.getOp().accept(this);
        prefixExpression.getExpression().accept(this);
      }

      public void visitAssignmentExpression(AssignmentExpression
                                              assignmentExpression) {
        assignmentExpression.getLeftHandSide().accept(this);
        result.append(" ");
        assignmentExpression.getOp().accept(this);
        result.append(" ");
        assignmentExpression.getRightHandSide().accept(this);
      }

      public void visitClassInstanceCreation(ClassInstanceCreation
                                               classInstanceCreation)
                                                   {
```

```
330       if (classInstanceCreation.getExpression() != null) {
            classInstanceCreation.getExpression().accept(this);
            result.append(" ");
          }
          result.append("new ");
335       classInstanceCreation.getType().accept(this);
          result.append("(");
          for (Expression e : classInstanceCreation.getArgumentList().
              getList()) {
            e.accept(this);
            result.append(", ");
340       }
          if (!classInstanceCreation.getArgumentList().getList().
              isEmpty()) {
            result.delete(result.length() - 2, result.length());
          }
          result.append(")");
345     }

        public void visitCastExpression(CastExpression castExpression)
            {
          result.append("(");
          castExpression.getType().accept(this);
350       result.append(") ");
          castExpression.getExpression().accept(this);
        }

        public void visitParentherizedExpression(
            ParentherizedExpression
355                                             parentherizedExpression
                                                    ) {
          result.append("(");
          parentherizedExpression.getExpression().accept(this);
          result.append(")");
        }
360
        public void visitFieldAccessExpression(FieldAccessExpression
                                               fieldAccessExpression)
                                                   {
          if (fieldAccessExpression.getExpression() != null) {
            fieldAccessExpression.getExpression().accept(this);
365         result.append(".");
          }
          fieldAccessExpression.getSimpleName().accept(this);
        }

370     public void visitThisExpression(ThisExpression thisExpression)
            {
          if (thisExpression.getSimpleName() != null) {
```

```
      thisExpression.getSimpleName().accept(this);
      result.append(".");
    }
    result.append("this");
  }

  public void visitArrayCreation(ArrayCreation arrayCreation) {
    result.append("new ");
    arrayCreation.getType().accept(this);
    result.append("[");
    if (arrayCreation.getExpression() != null) {
      arrayCreation.getExpression().accept(this);
    }
    result.append("]");
    if (arrayCreation.getExpressionList() != null) {
      result.append("{");
      for (Expression expression : arrayCreation.
          getExpressionList().getList()) {
        expression.accept(this);
        result.append(", ");
      }
      if (!arrayCreation.getExpressionList().getList().isEmpty()
          ) {
        result.delete(result.length() - 2, result.length());
      }
      result.append("}");
    }
  }

  public void visitVariableDeclarationExpression(
      VariableDeclarationExpression
                                                    variableDeclarationExpression
                                                        ) {
    result.append("VISITING VARIABLEDECLARATION_EXPRESSION");
  }

  public void visitBooleanLiteral(BooleanLiteral booleanLiteral)
      {
    result.append(booleanLiteral.getText());
  }

  public void visitIntegerLiteral(IntegerLiteral integerLiteral)
      {
    result.append(integerLiteral.getText());
  }

  public void visitDoubleLiteral(DoubleLiteral doubleLiteral) {
    result.append(doubleLiteral.getText());
  }
```

```
415
    public void visitCharLiteral(CharLiteral charLiteral) {
      result.append("\'" + charLiteral.getText() + "\'");
    }

420 public void visitStringLiteral(StringLiteral stringLiteral) {
      result.append("\"" + stringLiteral.getText() + "\"");
    }

    public void visitNullLiteral(NullLiteral nullLiteral) {
425   result.append("null");
    }

    //Other
    public void visitQualifiedName(QualifiedName qualifiedName) {
430   qualifiedName.getLeft().accept(this);
      result.append(".");
      qualifiedName.getRight().accept(this);
    }

435 public void visitSimpleName(SimpleName simpleName) {
      result.append(simpleName.getText());
    }

    public void visitInfixOperator(InfixOperator infixOperator) {
440   result.append(infixOperator.getText());
    }

    public void visitPrefixOperator(PrefixOperator prefixOperator)
        {
      result.append(prefixOperator.getText());
445 }

    public void visitAssignmentOperator(AssignmentOperator
       assignmentOperator) {
      result.append(assignmentOperator.getText());
    }
450
    public void visitModifier(Modifier modifier) {
      result.append(modifier.getText());
    }

455 public String result() {
      return result.toString();
    }
  }
```

E.5.8 StringVisitor.java

```
package translator.lib;

import translator.rslast.*;

public class StringVisitor
    extends Visitor {
  private StringBuffer result;

  public StringVisitor() {
    this.result = new StringBuffer();
  }

  public void visitRSLAst(RSLAst rslast) {
    super.visitRSLAst(rslast);
  }

  public void visitLibModule(LibModule module) {
    for (Id id : module.getContextList().getList()) {
      id.accept(this);
      result.append(", ");
    }
    if (!module.getContextList().getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
      result.append("\n");
    }
    module.getSchemeDef().accept(this);
  }

  public void visitSchemeDef(SchemeDef scheme) {
    result.append("scheme ");
    scheme.getId().accept(this);
    result.append(" =\n");
    scheme.getClassExpr().accept(this);
  }

  public void visitExtendingClassExpr(ExtendingClassExpr
      extendingClassExpr) {
    result.append(" extend ");
    extendingClassExpr.getBaseClass().accept(this);
    result.append(" with ");
    extendingClassExpr.getExtensionClass().accept(this);
  }

  public void visitSchemeInstantiation(SchemeInstantiation
      schemeInstantiation) {
    schemeInstantiation.getId().accept(this);
  }

  public void visitBasicClassExpr(BasicClassExpr basicClassExpr)
```

```
        {
      result.append("class\n");
      for (Decl decl : basicClassExpr.getDeclList().getList()) {
50      decl.accept(this);
      }
      result.append("end");
    }

55  /*Type Declarations*/
    public void visitTypeDecl(TypeDecl typeDecl) {
      result.append("type\n");
      for (TypeDef typeDef : typeDecl.getTypeDefList().getList())
          {
        typeDef.accept(this);
60      result.append(",\n");
      }
      if (!typeDecl.getTypeDefList().getList().isEmpty()) {
        result.delete(result.length() - 2, result.length());
      }
65    result.append("\n");
    }

    public void visitSortDef(SortDef sortDef) {
      sortDef.getId().accept(this);
70  }

    public void visitVariantDef(VariantDef variantDef) {
      variantDef.getId().accept(this);
      result.append(" == ");
75    for (Variant variant : variantDef.getVariantList().getList()
         ) {
        variant.accept(this);
        result.append(" | ");
      }
      if (!variantDef.getVariantList().getList().isEmpty()) {
80      result.delete(result.length() - 3, result.length());
      }
    }

    public void visitShortRecordDef(ShortRecordDef shortRecordDef)
        {
85    shortRecordDef.getId().accept(this);
      result.append(" :: ");
      for (ComponentKind componentKind :
           shortRecordDef.getComponentKindString().getList()) {
        componentKind.accept(this);
90      result.append(" ");
      }
      if (!shortRecordDef.getComponentKindString().getList().
```

```
        isEmpty()) {
      result.delete(result.length() - 1, result.length());
    }
  }

  public void visitConstructor(Constructor constructor) {
    constructor.getId().accept(this);
  }

  public void visitDestructor(Destructor destructor) {
    destructor.getId().accept(this);
  }

  public void visitRecordVariant(RecordVariant recordVariant) {
    recordVariant.getConstructor().accept(this);
    result.append("(");
    for (ComponentKind componentKind :
        recordVariant.getComponentKindList().getList()) {
      componentKind.accept(this);
      result.append(", ");
    }
    if (!recordVariant.getComponentKindList().getList().isEmpty
        ()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")");
  }

  public void visitComponentKind(ComponentKind componentKind) {
    if (componentKind.getOptionalDestructor() != null) {
      componentKind.getOptionalDestructor().accept(this);
      result.append(" : ");
    }
    componentKind.getTypeExpr().accept(this);
    if (componentKind.getOptionalReconstructor() != null) {
      result.append(" <-> ");
      componentKind.getOptionalReconstructor().accept(this);
    }
  }

  /*Value Declarations*/
  public void visitValueDecl(ValueDecl valueDecl) {
    result.append("value\n");
    for (ValueDef valueDef : valueDecl.getValueDefList().getList
        ()) {
      valueDef.accept(this);
      result.append(",\n");
    }
    if (!valueDecl.getValueDefList().getList().isEmpty()) {
```

```
      result.delete(result.length() - 2, result.length());
    }
    result.append("\n");
  }

  public void visitExplicitFunctionDef(ExplicitFunctionDef
      explicitFunctionDef) {
    explicitFunctionDef.getSingleTyping().accept(this);
    explicitFunctionDef.getFormalFunctionApplication().accept(
        this);
    result.append(" is ");
    explicitFunctionDef.getValueExpr().accept(this);
    explicitFunctionDef.getOptionalPrecondition().accept(this);
  }

  public void visitSingleTyping(SingleTyping singleTyping) {
    singleTyping.getBinding().accept(this);
    result.append(" : ");
    singleTyping.getTypeExpr().accept(this);
    result.append("\n");
  }

  public void visitIdApplication(IdApplication idApplication) {
    idApplication.getId().accept(this);
    result.append("(");
    boolean hasParameters = false;
    for (FormalFunctionParameter parameter :
        idApplication.getFormalFunctionParameters().getList())
            {
      parameter.accept(this);
      result.append(", ");
      hasParameters = true;
    }
    if (hasParameters) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")");
  }

  public void visitFormalFunctionParameter(
      FormalFunctionParameter
                                          formalFunctionParameter
                                              ) {
    boolean hasElements = false;
    for (Binding binding : formalFunctionParameter.
        getBindingList().getList()) {
      binding.accept(this);
      result.append(", ");
      hasElements = true;
```

```
    }
    if (hasElements) {
      result.delete(result.length() - 2, result.length() - 0);
    }
  }

  public void visitPrecondition(Precondition precondition) {
    result.append("\nPRECONDITION ");
  }

  public void visitNoPrecondition(NoPrecondition noPrecondition)
      {
  }

  //Test Declarations
  public void visitTestDecl(TestDecl testDecl) {
    result.append("test_case\n");
    for (TestDef testDef : testDecl.getTestDefList().getList())
        {
      testDef.accept(this);
      result.append(",\n");
    }
    if (!testDecl.getTestDefList().getList().isEmpty()) {
      result.delete(result.length() - 2, result.length());
    }
    result.append("\n");

  }

  public void visitTestDef(TestDef testDef) {
    result.append("[");
    testDef.getId().accept(this);
    result.append("] ");
    testDef.getValueExpr().accept(this);
  }

  //Type Expressions
  public void visitFunctionTypeExpr(FunctionTypeExpr
      functionTypeExpr) {
    super.visitFunctionTypeExpr(functionTypeExpr);
  }

  public void visitFunctionArrow(FunctionArrow functionArrow) {
    result.append(" ");
    result.append(functionArrow.getText());
    result.append(" ");
  }

  public void visitFunctionResultDescription(
```

```
      FunctionResultDescription
                                                  functionResultDescription
                                                      ) {
    super.visitFunctionResultDescription(
        functionResultDescription);
  }

  public void visitNoAccessDescription(NoAccessDescription
      noAccessDescription) {}

  public void visitFiniteListTypeExpr(FiniteListTypeExpr
      finiteListExpr) {
    finiteListExpr.getTypeExpr().accept(this);
    result.append("*");
  }

  public void visitProductTypeExpr(ProductTypeExpr
      productTypeExpr) {
    result.append("(");
    boolean hasElements = false;
    for (TypeExpr te : productTypeExpr.getTypeExprList().getList
        ()) {
      te.accept(this);
      result.append(" >< ");
      hasElements = true;
    }
    if (hasElements) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")");
  }

  public void visitTypeLiteral(TypeLiteral typeLiteral) {
    result.append(typeLiteral.getText());
  }

  public void visitTypeName(TypeName typeName) {
    typeName.getId().accept(this);
  }

  //Value Expressions
  public void visitApplicationExpr(ApplicationExpr
      applicationExpr) {
    applicationExpr.getValueExpr().accept(this);
    result.append("(");
    boolean hasElements = false;
    for (ValueExpr ve : applicationExpr.getOptionalValueExprList
        ().getList()) {
      ve.accept(this);
```

```
        result.append(", ");
        hasElements = true;
      }
      if (hasElements) {
        result.delete(result.length() - 2, result.length());
      }
      result.append(")");
    }

    public void visitEnumeratedListExpr(EnumeratedListExpr
        enumeratedListExpr) {
      result.append("<.");
      boolean hasElements = false;
      for (ValueExpr ve : enumeratedListExpr.getValueExprList().
          getList()) {
        ve.accept(this);
        result.append(", ");
        hasElements = true;
      }
      if (hasElements) {
        result.delete(result.length() - 2, result.length());
      }
      result.append(".>");
    }

    public void visitIfExpr(IfExpr ifExpr) {
      result.append("if ");
      ifExpr.getCondition().accept(this);
      result.append(" then ");
      ifExpr.getValueExpr().accept(this);
      for (ElsifBranch elsif : ifExpr.getElsifBranchList().getList
          ()) {
        elsif.accept(this);
      }
      if (ifExpr.getElseBranch() != null) {
        ifExpr.getElseBranch().accept(this);
      }
      result.append(" end");
    }

    public void visitElsifBranch(ElsifBranch elsifBranch) {
      result.append(" elsif ");
      elsifBranch.getCondition().accept(this);
      result.append(" then ");
      elsifBranch.getValueExpr().accept(this);
    }

    public void visitElseBranch(ElseBranch elseBranch) {
      result.append(" else ");
```

```
      elseBranch.getValueExpr().accept(this);
315   }

      public void visitCaseExpr(CaseExpr caseExpr) {
        result.append("case ");
        caseExpr.getCondition().accept(this);
320     result.append(" of \n");
        for (CaseBranch cb : caseExpr.getCaseBranchList().getList())
            {
          cb.accept(this);
          result.append(",\n");
        }
325     if (!caseExpr.getCaseBranchList().getList().isEmpty()) {
          result.delete(result.length() - 2, result.length());
        }
        result.append("\nend");
      }
330
      public void visitCaseBranch(CaseBranch caseBranch) {
        caseBranch.getPattern().accept(this);
        result.append(" -> ");
        caseBranch.getValueExpr().accept(this);
335   }

      public void visitNamePattern(NamePattern namePattern) {
        namePattern.getId().accept(this);
      }
340
      public void visitRecordPattern(RecordPattern recordPattern) {
        recordPattern.getValueOrVariableName().accept(this);
        result.append("(");
        for (Pattern p : recordPattern.getInnerPatternList().getList
            ()) {
345       p.accept(this);
          result.append(", ");
        }
        if (!recordPattern.getInnerPatternList().getList().isEmpty()
            ) {
          result.delete(result.length() - 2, result.length());
350     }
        result.append(")");
      }

      public void visitValueLiteralPattern(ValueLiteralPattern
          valueLiteralPattern) {
355     valueLiteralPattern.getValueLiteral().accept(this);
      }

      public void visitWildcardPattern(WildcardPattern
```

```
      wildcardPattern) {
    result.append("_");
  }

  public void visitDisambiguationExpr(DisambiguationExpr
      disambiguationExpr) {
    disambiguationExpr.getValueExpr().accept(this);
    result.append(" : ");
    disambiguationExpr.getTypeExpr().accept(this);
  }

  public void visitValueInfixExpr(ValueInfixExpr valueInfixExpr)
      {
    valueInfixExpr.getLeft().accept(this);
    result.append(" ");
    valueInfixExpr.getOp().accept(this);
    result.append(" ");
    valueInfixExpr.getRight().accept(this);
  }

  public void visitValuePrefixExpr(ValuePrefixExpr
      valuePrefixExpr) {
    result.append(" ");
    valuePrefixExpr.getOp().accept(this);
    result.append(" ");
    valuePrefixExpr.getValueExpr().accept(this);
  }

  public void visitRSLInfixOp(RSLInfixOp rslInfixOp) {
    result.append(rslInfixOp.getText());
  }

  public void visitRSLPrefixOp(RSLPrefixOp rslPrefixOp) {
    result.append(rslPrefixOp.getText());
  }

  public void visitValueOrVariableName(ValueOrVariableName
      valueOrVariableName) {
    valueOrVariableName.getId().accept(this);
  }

  public void visitValueLiteralInteger(ValueLiteralInteger
      valueLiteralInteger) {
    result.append(valueLiteralInteger.getText());
  }

  public void visitValueLiteralReal(ValueLiteralReal
      valueLiteralReal) {
    result.append(valueLiteralReal.getText());
```

```
  }

  public void visitValueLiteralBool(ValueLiteralBool
      valueLiteralBool) {
    result.append(valueLiteralBool.getText());
  }

  public void visitValueLiteralText(ValueLiteralText
      valueLiteralText) {
    result.append("\"" + valueLiteralText.getText() + "\"");
  }

  public void visitValueLiteralChar(ValueLiteralChar
      valueLiteralChar) {
    result.append("\'" + valueLiteralChar.getText() + "\'");
  }

  /*Common RSL*/
  public void visitId(Id id) {
    result.append(id.getText());
  }

  /*Auxillary functions*/
  public String result() {
    return result.toString();
  }
}
```

E.5.9 TypeDecorateVisitor.java

```
package translator.lib;

import translator.javaast.*;
import translator.rslast.*;
import translator.rsllib.*;

import java.util.*;

public class TypeDecorateVisitor
    extends Visitor {
  private HashMap<Id, HashMap<Id, TypeEvaluator>> map;
  private HashMap<Id, TypeEvaluator> currentMap;
  private ArrayList<TypeEvaluator> currentRecordType;
  private ArrayList<Id> currentRecordId;
  private HashMap<Id, ArrayList<TypeEvaluator>>
      recordMapType;
  private HashMap<Id, ArrayList<Id>> recordMapId;
  private HashMap<Id, ArrayList<TypeEvaluator>>
      functionMapType;
```

```
  private HashMap < Id , TypeEvaluator > functionResultType;
  private HashMap < Id , Id > parentMap;
  private ArrayList < String > getMethodList;

  public TypeDecorateVisitor() {
    this.map = new HashMap < Id , HashMap < Id , TypeEvaluator
        >> ();
    this.recordMapType = new HashMap < Id , ArrayList <
        TypeEvaluator >> ();
    this.recordMapId = new HashMap < Id , ArrayList < Id >> ();
    this.functionMapType = new HashMap < Id , ArrayList <
        TypeEvaluator >> ();
    this.functionResultType = new HashMap < Id , TypeEvaluator
        > ();
    this.parentMap = new HashMap < Id , Id > ();
    this.getMethodList = new ArrayList < String > ();
  }

  ///*
   public TypeDecorateVisitor(HashMap < Id , HashMap < Id ,
       TypeEvaluator >> map,
                              HashMap < Id ,
                              ArrayList < TypeEvaluator >>
                                  recordMapType,
                              HashMap < Id , ArrayList < Id >>
                                  recordMapId,
                              HashMap < Id ,
                              ArrayList < TypeEvaluator >>
                                  functionMapType,
                              HashMap < Id , TypeEvaluator >
                                  functionResultType,
                              HashMap < Id , Id > parentMap,
                              ArrayList < String > getMethodList
                                  ) {
    this.map = map;
    this.recordMapType = recordMapType;
    this.recordMapId = recordMapId;
    this.functionMapType = functionMapType;
    this.functionResultType = functionResultType;
    this.parentMap = parentMap;
    this.getMethodList = getMethodList;
   }

  //*/

  public HashMap < Id , HashMap < Id , TypeEvaluator >> getMap() {
    return map;
  }
```

```
  public void visitRSLAst(RSLAst rslast) {
    rslast.getLibModule().accept(this);
  }

  public void visitLibModule(LibModule module) {
    module.getSchemeDef().accept(this);
  }

  public void visitSchemeDef(SchemeDef scheme) {
    scheme.getClassExpr().accept(this);
  }

  public void visitBasicClassExpr(BasicClassExpr basicClassExpr)
      {
    for (Decl decl : basicClassExpr.getDeclList().getList()) {
      decl.accept(this);
    }
  }

  /*Type Declarations*/
  public void visitTypeDecl(TypeDecl typeDecl) {
    for (TypeDef typeDef : typeDecl.getTypeDefList().getList())
        {
      typeDef.accept(this);
    }
  }

  public void visitSortDef(SortDef sortDef) {
    sortDef.getId().accept(this);
  }

  public void visitVariantDef(VariantDef variantDef) {
    variantDef.getId().accept(this);
    for (Variant variant : variantDef.getVariantList().getList()
        ) {
      variant.accept(this);
    }
  }

  public void visitShortRecordDef(ShortRecordDef shortRecordDef)
      {
    this.currentMap = new HashMap < Id, TypeEvaluator > ();
    this.currentRecordType = new ArrayList < TypeEvaluator > ();
    this.currentRecordId = new ArrayList < Id > ();

    shortRecordDef.getId().accept(this);
    for (ComponentKind componentKind :
        shortRecordDef.getComponentKindString().getList()) {
      componentKind.accept(this);
```

```
    }

    this.currentMap.remove(shortRecordDef.getId());
    this.currentMap.put(shortRecordDef.getId(), shortRecordDef);
    //this.currentMap.put(shortRecordDef.getId(), new TypeName(
        shortRecordDef.getId()));

    this.parentMap.put(shortRecordDef.getId(), shortRecordDef.
        getId());
    this.recordMapType.put(shortRecordDef.getId(), this.
        currentRecordType);
    this.recordMapId.put(shortRecordDef.getId(), this.
        currentRecordId);
    this.map.put(shortRecordDef.getId(), this.currentMap);
  }

  public void visitConstructor(Constructor constructor) {
    if (!(constructor.getParent() instanceof RecordVariant)) {
      this.currentMap = new HashMap<Id, TypeEvaluator>();
    }

    constructor.getId().accept(this);
    this.currentMap.remove(constructor.getId());
    this.currentMap.put(constructor.getId(), constructor);
    //this.currentMap.put(constructor.getId(), new TypeName(
        constructor.getId()));

    if (constructor.getParent() instanceof VariantDef) {
      this.parentMap.put(constructor.getId(),
                         ((VariantDef) constructor.getParent())
                           .getId());
    }

    if (!(constructor.getParent() instanceof RecordVariant)) {
      this.map.put(constructor.getId(), this.currentMap);
    }
  }

  public void visitDestructor(Destructor destructor) {
    destructor.getId().accept(this);
  }

  public void visitRecordVariant(RecordVariant recordVariant) {
    this.currentMap = new HashMap<Id, TypeEvaluator>();
    this.currentRecordType = new ArrayList<TypeEvaluator>();
    this.currentRecordId = new ArrayList<Id>();

    recordVariant.getConstructor().accept(this);
    for (ComponentKind componentKind :
```

```
145           recordVariant.getComponentKindList().getList()) {
        componentKind.accept(this);
      }

      //System.out.println("Parent: " + recordVariant.getParent())
          ;
150
      this.parentMap.put(recordVariant.getConstructor().getId(),
                         ((VariantDef) recordVariant.getParent())
                             .getId());
      this.recordMapType.put(recordVariant.getConstructor().getId
          (),
                             this.currentRecordType);
155   this.recordMapId.put(recordVariant.getConstructor().getId(),
                           this.currentRecordId);
      this.map.put(recordVariant.getConstructor().getId(), this.
          currentMap);
    }

160 public void visitComponentKind(ComponentKind componentKind) {
      componentKind.getTypeExpr().accept(this);
      if (componentKind.getOptionalDestructor() != null) {
        componentKind.getOptionalDestructor().accept(this);
        this.currentMap.remove(componentKind.getOptionalDestructor
            ().getId());
165     this.currentMap.put(componentKind.getOptionalDestructor().
            getId(),
                            componentKind.getOptionalDestructor())
                                ;
        this.currentRecordType.add(componentKind.getTypeExpr());
        this.currentRecordId.add(componentKind.
            getOptionalDestructor().getId());
        //Added
170     this.map.put(componentKind.getOptionalDestructor().getId()
            ,
                     this.currentMap);
      }
      if (componentKind.getOptionalReconstructor() != null) {
        componentKind.getOptionalReconstructor().accept(this);
175   }
    }

    /*Value Declarations*/
    public void visitValueDecl(ValueDecl valueDecl) {
180   for (ValueDef valueDef : valueDecl.getValueDefList().getList
          ()) {
      valueDef.accept(this);
      }
    }
```

```
185   public void visitExplicitFunctionDef(ExplicitFunctionDef
          explicitFunctionDef) {
        this.currentMap = new HashMap < Id, TypeEvaluator > ();
        this.currentRecordType = new ArrayList < TypeEvaluator > ();

        explicitFunctionDef.getSingleTyping().accept(this);
190     Id functionId = (Id) explicitFunctionDef.getSingleTyping().
            getBinding();
        explicitFunctionDef.getFormalFunctionApplication().accept(
            this);
        explicitFunctionDef.getOptionalPrecondition().accept(this);

        RSLList < TypeExpr > parameterList = null;
195     if (explicitFunctionDef.getSingleTyping().getTypeExpr()
            instanceof
            FunctionTypeExpr) {
          TypeExpr parameters = ( (FunctionTypeExpr)
              explicitFunctionDef.
                                  getSingleTyping().getTypeExpr()).
                                      getTypeExpr();
          parameterList = new RSLListDefault < TypeExpr > ();
200       if (parameters instanceof ProductTypeExpr) {
            ProductTypeExpr pte = (ProductTypeExpr) parameters;
            for (TypeExpr te : pte.getTypeExprList().getList()) {
              parameterList.getList().add(te);
              this.currentRecordType.add(te);
205         }
          }
          else if (parameters instanceof ListTypeExpr) {
            parameterList.getList().add(parameters);
            this.currentRecordType.add(parameters);
210       }
          else if (parameters instanceof TypeLiteral) {
            parameterList.getList().add(parameters);
            this.currentRecordType.add(parameters);
          }
215       else if (parameters instanceof TypeName) {
            parameterList.getList().add(parameters);
            this.currentRecordType.add(parameters);
          }
        }
220     if (explicitFunctionDef.getFormalFunctionApplication()
            instanceof
            IdApplication) {
          IdApplication ia = (IdApplication) explicitFunctionDef.
              getFormalFunctionApplication();
          for (FormalFunctionParameter ffp :
225             ia.getFormalFunctionParameters().getList()) {
```

```
            for (Binding b : ffp.getBindingList().getList()) {
              currentMap.put( ( (Id) b), parameterList.hd());
              parameterList = parameterList.tl();
            }
          }
        }
        this.map.put(functionId , currentMap);
        this.functionMapType.put(functionId , this.currentRecordType)
            ;

        explicitFunctionDef.setTypeEvaluatorForTypeEvaluation( ( (
            FunctionTypeExpr)
             explicitFunctionDef.getSingleTyping().getTypeExpr()).
             getFunctionResultDescription().getTypeExpr());
        explicitFunctionDef.getValueExpr().accept(this);
    }

    public void visitSingleTyping(SingleTyping singleTyping) {
        singleTyping.getBinding().accept(this);
        singleTyping.getTypeExpr().accept(this);
        this.functionResultType.put( (Id) singleTyping.getBinding(),
                                    ( (FunctionTypeExpr)
                                        singleTyping.getTypeExpr()).
                                    getFunctionResultDescription().
                                        getTypeExpr());
    }

    public void visitIdApplication(IdApplication idApplication) {
        idApplication.getId().accept(this);
        for (FormalFunctionParameter parameter :
             idApplication.getFormalFunctionParameters().getList())
                 {
          parameter.accept(this);
        }
    }

    public void visitFormalFunctionParameter(
        FormalFunctionParameter
                                                formalFunctionParameter
                                                    ) {
        for (Binding binding : formalFunctionParameter.
            getBindingList().getList()) {
          binding.accept(this);
        }
    }

    public void visitPrecondition(Precondition precondition) {}

    public void visitNoPrecondition(NoPrecondition noPrecondition)
```

```
    {}

  //Type Expressions
  public void visitFunctionTypeExpr(FunctionTypeExpr
      functionTypeExpr) {
    functionTypeExpr.getTypeExpr().accept(this);
    functionTypeExpr.getFunctionArrow().accept(this);
    functionTypeExpr.getFunctionResultDescription().accept(this)
        ;
  }

  public void visitFunctionArrow(FunctionArrow functionArrow) {}

  public void visitFunctionResultDescription(
      FunctionResultDescription
                                              functionResultDescription
                                                  ) {
    functionResultDescription.getOptionalAccessDescription().
        accept(this);
    functionResultDescription.getTypeExpr().accept(this);
  }

  public void visitNoAccessDescription(NoAccessDescription
      noAccessDescription) {}

  public void visitFiniteListTypeExpr(FiniteListTypeExpr
      finiteListExpr) {
    finiteListExpr.getTypeExpr().accept(this);
  }

  public void visitProductTypeExpr(ProductTypeExpr
      productTypeExpr) {
    for (TypeExpr te : productTypeExpr.getTypeExprList().getList
        ()) {
      te.accept(this);
    }
  }

  public void visitTypeLiteral(TypeLiteral typeLiteral) {}

  public void visitTypeName(TypeName typeName) {}

  public void visitId(Id id) {
    if (currentMap != null && !currentMap.containsKey(id)) {
      currentMap.put(id, null);
    }
  }

  //Value Expressions
```

```
  public void visitApplicationExpr(ApplicationExpr
      applicationExpr) {
    applicationExpr.getValueExpr().accept(this);

    for (ValueExpr ve : applicationExpr.getOptionalValueExprList
        ().getList()) {
      ve.accept(this);
    }
    if (applicationExpr.getValueExpr() instanceof
        ValueOrVariableName) {
      ValueOrVariableName vovn = (ValueOrVariableName)
          applicationExpr.
          getValueExpr();
      Id id = vovn.getId();

      //In case of a short record definition make function we
          need to lookup the short record definition
      if (map.get(id) == null) {
        if (vovn.getId().getText().startsWith("mk_")) {
          if (map.get(new Id(vovn.getId().getText().substring(3)
              )) != null) {
            id = new Id(vovn.getId().getText().substring(3));
            //System.out.println("Must be a
                shortRecordDefinition: " + id + " to: " + (map.
                get(id)).get(id));
            applicationExpr.setTypeEvaluatorForTypeEvaluation( (
                map.get(id)).
                get(id));

            ArrayList < TypeEvaluator > typeList = new ArrayList
                <
                TypeEvaluator > ();
            typeList = this.recordMapType.get(id);

            int i = 0;
            for (ValueExpr ve :
                 applicationExpr.getOptionalValueExprList().
                     getList()) {
              if (ve.getTypeEvaluatorForTypeEvaluation() == null
                  ) {
                ve.setTypeEvaluatorForTypeEvaluation(typeList.
                    get(i));
                //System.out.println("Setting ve: " + ve
                    + " : " + typeList.get(i));
              }
              i++;
            }
          }
        }
```

```
      else if (getMethodList.contains(id.getText())) {
        System.out.println(
            "Must be an application of an extra inserted
                getMethod");
        applicationExpr.setTypeEvaluatorForTypeEvaluation(new
            Destructor(id));
      }
    }
    else if (map.get(id) != null && this.functionMapType.get(
        id) == null) {
      /*Record variant*/
      //System.out.println("Must be a record variant or
          destructor: " + id + " to: " + (map.get(id)).get(id))
          ;

      applicationExpr.setTypeEvaluatorForTypeEvaluation( (map.
          get(id)).get(id));

      if (this.recordMapType.get(id) != null) {
        ArrayList < TypeEvaluator > typeList = new ArrayList
            < TypeEvaluator > ();
        typeList = this.recordMapType.get(id);

        int i = 0;
        for (ValueExpr ve :
             applicationExpr.getOptionalValueExprList().
                 getList()) {
          if (ve.getTypeEvaluatorForTypeEvaluation() == null)
              {
            ve.setTypeEvaluatorForTypeEvaluation(typeList.get(
                i));
            //System.out.println("Setting ve: " + ve
                + " : " + typeList.get(i));
          }
          i++;
        }
      }
    }
    else {
      //System.out.println("Must be an application of a
          function: " + id + " to: " + (map.get(id)).get(id));
      //System.out.println("FunctionResultType: " +
          functionResultType);
      ArrayList < TypeEvaluator > typeList = new ArrayList <
          TypeEvaluator > ();
      typeList = this.functionMapType.get(id);

      applicationExpr.setTypeEvaluatorForTypeEvaluation(new
          FunctionResultDescription(null,
```

```
                                                (TypeExpr)
                                                    functionResultType.get(
                                                    id)));

            int i = 0;
            for (ValueExpr ve : applicationExpr.
                getOptionalValueExprList().getList()) {
              if (ve.getTypeEvaluatorForTypeEvaluation() == null) {
                ve.setTypeEvaluatorForTypeEvaluation(typeList.get(i)
                    );
                //System.out.println("Setting ve: " + ve + " : " +
                    typeList.get(i));
              }
              i++;
            }
          }
        }
      }

      public void visitEnumeratedListExpr(EnumeratedListExpr
          enumeratedListExpr) {
        for (ValueExpr ve : enumeratedListExpr.getValueExprList().
            getList()) {
          ve.accept(this);
        }
        if (!enumeratedListExpr.getValueExprList().getList().isEmpty
            ()) {
          //Setting type of expression via content inside of list

          TypeEvaluator te = enumeratedListExpr.getValueExprList().
              hd().
               getTypeEvaluatorForTypeEvaluation();
          TypeExpr type_expr = null;
          if (te instanceof ShortRecordDef) {
            type_expr = new TypeName( ( (ShortRecordDef) te).getId()
                );
          }
          else if (te instanceof Constructor) {
            type_expr = new TypeName(parentMap.get( ( (Constructor)
                te).getId()));
          }
          else if (te instanceof FunctionResultDescription) {
            type_expr = ( (FunctionResultDescription) te).
                getTypeExpr();
          }
          else {
            type_expr = (TypeExpr) te;
          }
          if (type_expr != null) {
```

```
            enumeratedListExpr.setTypeEvaluatorForTypeEvaluation(new
                FiniteListTypeExpr(type_expr));
415       }
        }
        else {
          //Setting type of expression via call to parent expression
          //System.out.println("enumeratedListExpr.getParent(): " +
              enumeratedListExpr.getParent());
420       if (enumeratedListExpr.getParent() != null) {
            if (enumeratedListExpr.getParent().
                getTypeEvaluatorForTypeEvaluation() != null) {
              if (! (enumeratedListExpr.getParent() instanceof
                  ApplicationExpr)) {
                //System.out.println("enumeratedListExpr.getParent()
                    .getTypeEvaluatorForTypeEvaluation(): " +
                    enumeratedListExpr.getParent().
                    getTypeEvaluatorForTypeEvaluation());
                enumeratedListExpr.setTypeEvaluatorForTypeEvaluation
                    (
425                  enumeratedListExpr.getParent().
                     getTypeEvaluatorForTypeEvaluation());
              }
            }
            else {
430           //System.out.println("Parent not type evaluated!!!");
            }
          }
        }
      }
435
      public void visitIfExpr(IfExpr ifExpr) {
        ifExpr.getCondition().accept(this);
        ifExpr.getValueExpr().accept(this);
        for (ElsifBranch elsif : ifExpr.getElsifBranchList().getList
            ()) {
440       elsif.accept(this);
        }
        ifExpr.getElseBranch().accept(this);

        //Get type of first branch and use it as type of whole expr
445     if (ifExpr.getValueExpr().getTypeEvaluatorForTypeEvaluation
            () != null) {
          ifExpr.setTypeEvaluatorForTypeEvaluation(ifExpr.
              getValueExpr().
              getTypeEvaluatorForTypeEvaluation());
          ifExpr.getElseBranch().setTypeEvaluatorForTypeEvaluation(
              ifExpr.
              getValueExpr().getTypeEvaluatorForTypeEvaluation());
450       //System.out.println("Setting type of IfExpr via if branch
```

```
            to: " + ifExpr.getTypeEvaluatorForTypeEvaluation() );
        }
        else if (ifExpr.getElseBranch().
            getTypeEvaluatorForTypeEvaluation() != null) {
          ifExpr.setTypeEvaluatorForTypeEvaluation(ifExpr.
              getElseBranch().
               getTypeEvaluatorForTypeEvaluation());
455       ifExpr.getValueExpr().setTypeEvaluatorForTypeEvaluation(
              ifExpr.
               getElseBranch().getTypeEvaluatorForTypeEvaluation());
          //System.out.println("Setting type of IfExpr via else
              branch: " + ifExpr.getValueExpr().
              getTypeEvaluatorForTypeEvaluation());
        }
        else {
460       //System.out.println("Type of if expression not set!!!");
        }
      }

      public void visitElsifBranch(ElsifBranch elsifBranch) {
465     elsifBranch.getCondition().accept(this);
        elsifBranch.getValueExpr().accept(this);
      }

      public void visitElseBranch(ElseBranch elseBranch) {
470     elseBranch.getValueExpr().accept(this);
        if (elseBranch.getValueExpr().
            getTypeEvaluatorForTypeEvaluation() != null) {
          elseBranch.setTypeEvaluatorForTypeEvaluation(elseBranch.
              getValueExpr().
               getTypeEvaluatorForTypeEvaluation());
          //System.out.println("Setting type of elsebranch to: " +
              elseBranch.getValueExpr().
              getTypeEvaluatorForTypeEvaluation());
475     }
      }

      public void visitValueInfixExpr(ValueInfixExpr valueInfixExpr)
          {
        valueInfixExpr.getLeft().accept(this);
480     valueInfixExpr.getOp().accept(this);
        valueInfixExpr.getRight().accept(this);

        if (valueInfixExpr.getLeft().
            getTypeEvaluatorForTypeEvaluation() != null) {
          valueInfixExpr.setTypeEvaluatorForTypeEvaluation(
              valueInfixExpr.getLeft().
485            getTypeEvaluatorForTypeEvaluation());
          if (valueInfixExpr.getRight().
```

```
            getTypeEvaluatorForTypeEvaluation() == null) {
          valueInfixExpr.getRight().
              setTypeEvaluatorForTypeEvaluation(
               valueInfixExpr.getLeft().
                   getTypeEvaluatorForTypeEvaluation());
        }
        else {

        }
      }
      else if (valueInfixExpr.getRight().
          getTypeEvaluatorForTypeEvaluation() != null) {
        valueInfixExpr.setTypeEvaluatorForTypeEvaluation(
            valueInfixExpr.getRight().
             getTypeEvaluatorForTypeEvaluation());
        if (valueInfixExpr.getLeft() == null) {
          valueInfixExpr.getLeft().
              setTypeEvaluatorForTypeEvaluation(
               valueInfixExpr.getRight().
                   getTypeEvaluatorForTypeEvaluation());
        }
      }
      else {
        //System.out.println("Type of ValueInfixExpr not set!!!");
      }
      //System.out.println("ValueInfixExpr.
          getTypeEvaluatorForTypeEvaluation(): " + valueInfixExpr.
          getTypeEvaluatorForTypeEvaluation());
    }

    public void visitValuePrefixExpr(ValuePrefixExpr
        valuePrefixExpr) {
      valuePrefixExpr.getOp().accept(this);
      valuePrefixExpr.getValueExpr().accept(this);

      //Get type of operator
      RSLPrefixOp op = valuePrefixExpr.getOp();
      if (op == RSLPrefixOp.RSL_PREFIX_OP_HD) {
        ValueExpr ve = valuePrefixExpr.getValueExpr();
        if (ve instanceof ValueOrVariableName) {
          ValueOrVariableName vovn = (ValueOrVariableName) ve;
          //HOPEFULLY THE TYPE OF THE VARIABLE OR VALUE IS A
              FINITELISTTYPEEXPR
          //IF NOT THERE MUST HAVE BEEN AN ERROR. WE SET THE TYPE
              OF THE EVALUATOR TO BE WHAT IS INSIDE THE LIST
          //System.out.println("CurrentMAP: " + currentMap);
          valuePrefixExpr.setTypeEvaluatorForTypeEvaluation( ( (
               FiniteListTypeExpr) currentMap.get(vovn.getId()))).
                   getTypeExpr());
```

```
        //valuePrefixExpr.setTypeEvaluatorForTypeEvaluation(
            currentMap.get(vovn.getId()));
      }
      else if (ve instanceof ListExpr) {
        ListExpr le = (ListExpr) valuePrefixExpr.getValueExpr()
            ;;
        if (le instanceof EnumeratedListExpr) {
          EnumeratedListExpr ele = (EnumeratedListExpr) le;
          valuePrefixExpr.setTypeEvaluatorForTypeEvaluation(ele.
              getTypeEvaluatorForTypeEvaluation());
        }
      }
    }
    else if (op == RSLPrefixOp.RSL_PREFIX_OP_TL) {
      ValueExpr ve = valuePrefixExpr.getValueExpr();
      if (ve instanceof ValueOrVariableName) {
        ValueOrVariableName vovn = (ValueOrVariableName) ve;
        //HOPEFULLY THE TYPE OF THE VARIABLE OR VALUE IS A
            FINITELISTTYPEEXPR
        valuePrefixExpr.setTypeEvaluatorForTypeEvaluation(
            currentMap.get(vovn.
             getId()));
      }
      else if (ve instanceof ListExpr) {
        ListExpr le = (ListExpr) valuePrefixExpr.getValueExpr()
            ;;
        if (le instanceof EnumeratedListExpr) {
          EnumeratedListExpr ele = (EnumeratedListExpr) le;
          valuePrefixExpr.setTypeEvaluatorForTypeEvaluation(new
              FiniteListTypeExpr( (TypeExpr) ele.
                                  getTypeEvaluatorForTypeEvaluation
                                      ()));
        }
      }
    }
  }

  public void visitCaseExpr(CaseExpr caseExpr) {
    caseExpr.getCondition().accept(this);
    for (CaseBranch cb : caseExpr.getCaseBranchList().getList())
        {
      cb.accept(this);
    }
    for (CaseBranch cb : caseExpr.getCaseBranchList().getList())
        {
      if (cb.getTypeEvaluatorForTypeEvaluation() != null) {
        caseExpr.setTypeEvaluatorForTypeEvaluation(cb.
            getTypeEvaluatorForTypeEvaluation());
      }
```

```
    }
    if (caseExpr.getTypeEvaluatorForTypeEvaluation() != null) {
      for (CaseBranch cb : caseExpr.getCaseBranchList().getList
          ()) {
        if (cb.getTypeEvaluatorForTypeEvaluation() == null) {
          cb.setTypeEvaluatorForTypeEvaluation(caseExpr.
              getTypeEvaluatorForTypeEvaluation());
        }
      }
    }
    else {
      if (caseExpr.getParent().getTypeEvaluatorForTypeEvaluation
          () != null) {
        caseExpr.setTypeEvaluatorForTypeEvaluation(caseExpr.
            getParent().
            getTypeEvaluatorForTypeEvaluation());
      }
      else {
        System.out.println("Cannot set type of case expr: " +
            caseExpr);
      }
    }
  }

  public void visitCaseBranch(CaseBranch caseBranch) {
    caseBranch.getPattern().accept(this);
    caseBranch.getValueExpr().accept(this);

    if (caseBranch.getValueExpr().
        getTypeEvaluatorForTypeEvaluation() != null) {
      caseBranch.setTypeEvaluatorForTypeEvaluation(caseBranch.
          getValueExpr().
          getTypeEvaluatorForTypeEvaluation());
    }
    else {
      //System.out.println("Cannot evaluate expression. Try
          setting via parent: " + caseBranch.getPattern());
      //Try set via parent
      if (caseBranch.getParent() != null) {
        if (caseBranch.getParent().
            getTypeEvaluatorForTypeEvaluation() != null) {
          caseBranch.setTypeEvaluatorForTypeEvaluation(
              caseBranch.getParent().
              getTypeEvaluatorForTypeEvaluation());
          //System.out.println("Setting via parent: " +
              caseBranch.getParent().
              getTypeEvaluatorForTypeEvaluation());
          caseBranch.getValueExpr().
              setTypeEvaluatorForTypeEvaluation(
```

```
                caseBranch.getParent().
                    getTypeEvaluatorForTypeEvaluation());
            }
            else {
              //System.out.println("Parent has no type");
            }
          }
          else {
            //System.out.println("Parent not set!!!");
          }
        }
      }

      public void visitNamePattern(NamePattern namePattern) {
        namePattern.getId().accept(this);
      }

      public void visitRecordPattern(RecordPattern recordPattern) {
        recordPattern.getValueOrVariableName().accept(this);

        ArrayList < TypeEvaluator >
            tl = recordMapType.get(recordPattern.
                getValueOrVariableName().getId());
        ArrayList < Id >
            il = recordMapId.get(recordPattern.
                getValueOrVariableName().getId());
        int i = 0;
        for (Pattern p : recordPattern.getInnerPatternList().getList
            ()) {
          p.accept(this);
          ((NamePattern) p).setTypeEvaluatorForTypeEvaluation(tl.
              get(i));
          currentMap.put( ((NamePattern) p).getId(), tl.get(i));

          ((NamePattern) p).setValueInitializer(il.get(i));
          i++;
        }
      }

      public void visitValueLiteralPattern(ValueLiteralPattern
          valueLiteralPattern) {
        valueLiteralPattern.getValueLiteral().accept(this);
      }

      public void visitWildcardPattern(WildcardPattern
          wildcardPattern) {}

      public void visitDisambiguationExpr(DisambiguationExpr
          disambiguationExpr) {
```

```
      disambiguationExpr.getValueExpr().accept(this);
      disambiguationExpr.getTypeExpr().accept(this);
    }

    public void visitRSLInfixOp(RSLInfixOp rslInfixOp) {}

    public void visitRSLPrefixOp(RSLPrefixOp rslPrefixOp) {}

    public void visitValueOrVariableName(ValueOrVariableName
        valueOrVariableName) {
      if (map != null && map.get(valueOrVariableName.getId()) !=
          null) {
        valueOrVariableName.setTypeEvaluatorForTypeEvaluation( (
            map.get(
             valueOrVariableName.getId())).get(valueOrVariableName.
                 getId()));
      }
      else if (currentMap != null && currentMap.get(
          valueOrVariableName.getId()) != null) {
        valueOrVariableName.setTypeEvaluatorForTypeEvaluation(
            currentMap.get(
             valueOrVariableName.getId()));
      }
    }

    public void visitValueLiteralInteger(ValueLiteralInteger
        valueLiteralInteger) {
      valueLiteralInteger.setTypeEvaluatorForTypeEvaluation(
          TypeLiteral.RSLINT);
    }

    public void visitValueLiteralReal(ValueLiteralReal
        valueLiteralReal) {
      valueLiteralReal.setTypeEvaluatorForTypeEvaluation(
          TypeLiteral.RSLREAL);
    }

    public void visitValueLiteralBool(ValueLiteralBool
        valueLiteralBool) {
      valueLiteralBool.setTypeEvaluatorForTypeEvaluation(
          TypeLiteral.RSLBOOL);
    }
  }
```

E.5.10 TypeEvaluator.java

```
package translator.lib;

import translator.rslast.*;
```

```
public interface TypeEvaluator extends Element{

}
```

E.5.11 Visitor.java

```
package translator.lib;

import translator.rslast.*;

public abstract class Visitor {
  public void visitRSLAst(RSLAst rslast) {
    rslast.getLibModule().accept(this);
  }

  public void visitLibModule(LibModule module) {
    module.getSchemeDef().accept(this);
    for (Id id : module.getContextList().getList()) {
      id.accept(this);
    }
  }

  public void visitSchemeDef(SchemeDef scheme) {
    scheme.getClassExpr().accept(this);
  }

  public void visitBasicClassExpr(BasicClassExpr basicClassExpr)
      {
    for (Decl decl : basicClassExpr.getDeclList().getList()) {
      decl.accept(this);
    }
  }

  public void visitExtendingClassExpr(ExtendingClassExpr
     extendingClassExpr) {
    extendingClassExpr.getBaseClass().accept(this);
    extendingClassExpr.getExtensionClass().accept(this);
  }

  public void visitSchemeInstantiation(SchemeInstantiation
     schemeInstantiation) {
    schemeInstantiation.getId().accept(this);
  }

  /*Type Declarations*/
  public void visitTypeDecl(TypeDecl typeDecl) {
    for (TypeDef typeDef : typeDecl.getTypeDefList().getList())
        {
```

```
      typeDef.accept(this);
    }
  }

  public void visitSortDef(SortDef sortDef) {
    sortDef.getId().accept(this);
  }

  public void visitVariantDef(VariantDef variantDef) {
    variantDef.getId().accept(this);
    for (Variant variant : variantDef.getVariantList().getList()
        ) {
      variant.accept(this);
    }
  }

  public void visitShortRecordDef(ShortRecordDef shortRecordDef)
      {
    shortRecordDef.getId().accept(this);
    for (ComponentKind componentKind :
        shortRecordDef.getComponentKindString().getList()) {
      componentKind.accept(this);
    }
  }

  public void visitConstructor(Constructor constructor) {
    constructor.getId().accept(this);
  }

  public void visitDestructor(Destructor destructor) {
    destructor.getId().accept(this);
  }

  public void visitRecordVariant(RecordVariant recordVariant) {
    recordVariant.getConstructor().accept(this);
    for (ComponentKind componentKind :
        recordVariant.getComponentKindList().getList()) {
      componentKind.accept(this);
    }
  }

  public void visitComponentKind(ComponentKind componentKind) {
    if (componentKind.getOptionalDestructor() != null) {
      componentKind.getOptionalDestructor().accept(this);
    }
    componentKind.getTypeExpr().accept(this);
    if (componentKind.getOptionalReconstructor() != null) {
      componentKind.getOptionalReconstructor().accept(this);
    }
```

```
  }

  /*Value Declarations*/
  public void visitValueDecl(ValueDecl valueDecl) {
    for (ValueDef valueDef : valueDecl.getValueDefList().getList
        ()) {
      valueDef.accept(this);
    }
  }

  public void visitExplicitFunctionDef(ExplicitFunctionDef
      explicitFunctionDef) {
    explicitFunctionDef.getSingleTyping().accept(this);
    explicitFunctionDef.getFormalFunctionApplication().accept(
        this);
    explicitFunctionDef.getValueExpr().accept(this);
    explicitFunctionDef.getOptionalPrecondition().accept(this);
  }

  public void visitSingleTyping(SingleTyping singleTyping) {
    singleTyping.getBinding().accept(this);
    singleTyping.getTypeExpr().accept(this);
  }

  public void visitIdApplication(IdApplication idApplication) {
    idApplication.getId().accept(this);
    for (FormalFunctionParameter parameter :
        idApplication.getFormalFunctionParameters().getList())
            {
      parameter.accept(this);
    }
  }

  public void visitFormalFunctionParameter(
      FormalFunctionParameter
                                          formalFunctionParameter
                                              ) {
    for (Binding binding : formalFunctionParameter.
        getBindingList().getList()) {
      binding.accept(this);
    }
  }

  public void visitPrecondition(Precondition precondition) {}

  public void visitNoPrecondition(NoPrecondition noPrecondition)
      {}

  //Test Declarations
```

```
  public void visitTestDecl(TestDecl testDecl) {
    for (TestDef testDef : testDecl.getTestDefList().getList())
        {
      testDef.accept(this);
130   }
  }

  public void visitTestDef(TestDef testDef) {
    testDef.getId().accept(this);
135   testDef.getValueExpr().accept(this);
  }

  //Type Expressions
  public void visitFunctionTypeExpr(FunctionTypeExpr
      functionTypeExpr) {
140   functionTypeExpr.getTypeExpr().accept(this);
    functionTypeExpr.getFunctionArrow().accept(this);
    functionTypeExpr.getFunctionResultDescription().accept(this)
        ;
  }

145 public void visitFunctionArrow(FunctionArrow functionArrow) {}

  public void visitFunctionResultDescription(
      FunctionResultDescription
                                              functionResultDescription
                                                 ) {
    functionResultDescription.getOptionalAccessDescription().
        accept(this);
150   functionResultDescription.getTypeExpr().accept(this);
  }

  public void visitNoAccessDescription(NoAccessDescription
      noAccessDescription) {}

155 public void visitFiniteListTypeExpr(FiniteListTypeExpr
      finiteListExpr) {
    finiteListExpr.getTypeExpr().accept(this);
  }

  public void visitProductTypeExpr(ProductTypeExpr
      productTypeExpr) {
160   for (TypeExpr te : productTypeExpr.getTypeExprList().getList
        ()) {
      te.accept(this);
    }
  }

165 public void visitTypeLiteral(TypeLiteral typeLiteral) {}
```

```
  public void visitTypeName(TypeName typeName) {
    typeName.getId().accept(this);
  }

  //Value Expressions
  public void visitApplicationExpr(ApplicationExpr
      applicationExpr) {
    applicationExpr.getValueExpr().accept(this);
    for (ValueExpr ve : applicationExpr.getOptionalValueExprList
        ().getList()) {
      ve.accept(this);
    }
  }

  public void visitEnumeratedListExpr(EnumeratedListExpr
      enumeratedListExpr) {
    for (ValueExpr ve : enumeratedListExpr.getValueExprList().
        getList()) {
      ve.accept(this);
    }
  }

  public void visitIfExpr(IfExpr ifExpr) {
    ifExpr.getCondition().accept(this);
    ifExpr.getValueExpr().accept(this);
    for (ElsifBranch elsif : ifExpr.getElsifBranchList().getList
        ()) {
      elsif.accept(this);
    }
    ifExpr.getElseBranch().accept(this);
  }

  public void visitElsifBranch(ElsifBranch elsifBranch) {
    elsifBranch.getCondition().accept(this);
    elsifBranch.getValueExpr().accept(this);
  }

  public void visitElseBranch(ElseBranch elseBranch) {
    elseBranch.getValueExpr().accept(this);
  }

  public void visitCaseExpr(CaseExpr caseExpr) {
    caseExpr.getCondition().accept(this);
    for (CaseBranch cb : caseExpr.getCaseBranchList().getList())
        {
      cb.accept(this);
    }
  }
```

```
210   public void visitCaseBranch(CaseBranch caseBranch) {
        caseBranch.getPattern().accept(this);
        caseBranch.getValueExpr().accept(this);
      }

215   public void visitNamePattern(NamePattern namePattern) {
        namePattern.getId().accept(this);
      }

      public void visitRecordPattern(RecordPattern recordPattern) {
220     recordPattern.getValueOrVariableName().accept(this);
        for (Pattern p : recordPattern.getInnerPatternList().getList
            ()) {
          p.accept(this);
        }
      }
225
      public void visitValueLiteralPattern(ValueLiteralPattern
          valueLiteralPattern) {
        valueLiteralPattern.getValueLiteral().accept(this);
      }

230   public void visitWildcardPattern(WildcardPattern
          wildcardPattern) {}

      public void visitDisambiguationExpr(DisambiguationExpr
          disambiguationExpr) {
        disambiguationExpr.getValueExpr().accept(this);
        disambiguationExpr.getTypeExpr().accept(this);
235   }

      public void visitValueInfixExpr(ValueInfixExpr valueInfixExpr)
          {
        valueInfixExpr.getLeft().accept(this);
        valueInfixExpr.getOp().accept(this);
240     valueInfixExpr.getRight().accept(this);
      }

      public void visitValuePrefixExpr(ValuePrefixExpr
          valuePrefixExpr) {
        valuePrefixExpr.getOp().accept(this);
245     valuePrefixExpr.getValueExpr().accept(this);
      }

      public void visitRSLInfixOp(RSLInfixOp rslInfixOp) {}

250   public void visitRSLPrefixOp(RSLPrefixOp rslPrefixOp) {}
```

```
    public void visitValueOrVariableName(ValueOrVariableName
        valueOrVariableName) {
      valueOrVariableName.getId().accept(this);
    }

    public void visitValueLiteralInteger(ValueLiteralInteger
        valueLiteralInteger) {}

    public void visitValueLiteralReal(ValueLiteralReal
        valueLiteralReal) {}

    public void visitValueLiteralBool(ValueLiteralBool
        valueLiteralBool) {}

    public void visitValueLiteralText(ValueLiteralText
        valueLiteralText) {}

    public void visitValueLiteralChar(ValueLiteralChar
        valueLiteralChar) {}

    /*Common*/
    public void visitId(Id id) {}
}
```

Appendix F

Source Code, Second Version

F.1 translator

F.1.1 Runner2.java

```
package translator;

import translator.lib.*;
import translator.rsllib.*;
import translator.rslast2.*;
import translator.javaast2.*;
import translator.syntacticanalyzer.*;

import java.util.*;
import java.io.*;

public class Runner2 {
    public static Runner2 instance;

    private Properties properties = null;

    private TM_OptionalPackageDeclaration opd;
    private TM_OptionalSimpleName ext;
    private TM_OptionalId visitorId;
    private RSLList<TM_ImportDeclaration> idl;
    private HashMap<String, HashMap<String, TM_TypeEvaluator>>
        map;
    private HashMap<String, ArrayList<TM_TypeEvaluator>>
        recordMapType;
    private HashMap<String, ArrayList<String>> recordMapId;
    private HashMap<String, ArrayList<TM_TypeEvaluator>>
        functionMapType;
    private HashMap<String, TM_TypeEvaluator> functionResultType
```

```
        ;
    private HashMap<String, String> parentMap;
    private ArrayList<String> ooExternalMethodList;
    private RSLList<String> ignoreList;
    private boolean writeExtensionFiles = false;

    public Runner2() {
        properties = new Properties();
        try {
            properties.load(new FileInputStream("default.
                properties"));
        }
        catch(FileNotFoundException fnfe) {
            System.out.println("default.properties file not
                found");
        }
        catch(IOException ioe) {
            System.out.println("IOException: " + ioe);
        }
        init();
    }

    public Runner2(String propertiesFileName) {
        properties = new Properties();
        try {
            properties.load(new FileInputStream(
                propertiesFileName));
        }
        catch(FileNotFoundException fnfe) {
            System.out.println(propertiesFileName + " file not
                found");
        }
        catch(IOException ioe) {
            System.out.println("IOException: " + ioe);
        }
        init();
    }

    public void init() {
        map = new HashMap<String, HashMap<String,
            TM_TypeEvaluator>>();
        recordMapType = new HashMap<String, ArrayList<
            TM_TypeEvaluator>>();
        recordMapId = new HashMap<String, ArrayList<String>>();
        functionMapType = new HashMap<String, ArrayList<
            TM_TypeEvaluator>>();
        functionResultType = new HashMap<String,
            TM_TypeEvaluator>();
        parentMap = new HashMap<String, String>();
```

```
        ooExternalMethodList = null;
        ignoreList = null;

        if(getProperty("getMethods") != null) {
70          ooExternalMethodList = new ArrayList<String>(Arrays.
                asList(getProperty("getMethods").split("$")));
        }
        else {
            ooExternalMethodList = new ArrayList<String>();
        }
75
        opd = null;
        if(getProperty("package") != null) {
            RSLList<String> t = new RSLListDefault<String>(
                getProperty("package").split("\\."));
            Collections.reverse(t.getList());
80          opd = new TM_PackageDeclaration(Translator_Module2.
                makeName(t));
        }
        else {
            opd = new TM_NoPackageDeclaration();
        }
85
        idl = new RSLListDefault<TM_ImportDeclaration>();
        idl.getList().add(
                        new TM_ImportDeclaration(
                                                    new Make_TM_SimpleName
                                                        (
90                      new TM_SimpleName("java.util.*")
                        )
                                                    ));
        idl.getList().add(
            new TM_ImportDeclaration(
95              new Make_TM_SimpleName(
                      new TM_SimpleName("translator.rsllib.*")
                      )
              ));
        if(getProperty("import") != null) {
100         idl.getList().addAll((Translator_Module2.
                makeImportDeclarationList(new RSLListDefault<
                String>(getProperty("import").split("\\$")))).
                getList());
        }

        ext = null;
        if(getProperty("extend") != null) {
105         ext = new Make_TM_OptionalSimpleName(new
                TM_SimpleName(getProperty("extend")));
        }
```

```
            else {
                ext = new TM_NoOptionalSimpleName();
            }

            visitorId = null;
            if(getProperty("visitor") != null) {
                visitorId = new Make_TM_Id(new TM_Id(getProperty("
                    visitor")));
            }
            else {
                visitorId = new TM_NoOptionalId();
            }

            if(getProperty("ignoreList") != null) {
                ignoreList = new RSLListDefault<String>(getProperty(
                    "ignoreList").split("\\$"));
                System.out.println("Ignoring: " + ignoreList);
            }
            else {
                ignoreList = new RSLListDefault<String>();
            }

            if((new Boolean(getProperty("printRSL"))).booleanValue()
                ) {
                writeExtensionFiles = true;
            }

        }

        public String getProperty(String key) {
            return this.properties.getProperty(key);
        }

        public void typeDecorate(TM_RSLAst rslast) {
            //System.out.println("Starting setting parent of
                expressions");
            TM_ParentRSLAstVisitor parentVisitor = new
                TM_ParentRSLAstVisitor();
            rslast.accept(parentVisitor);
            //System.out.println("Ending setting parent of
                expressions\n");

            TM_TypeDecorateRSLAstVisitor typeVisitor = new
                TM_TypeDecorateRSLAstVisitor(map, recordMapType,
                recordMapId, functionMapType, functionResultType,
                parentMap, ooExternalMethodList, false);
            rslast.accept(typeVisitor);
            TM_TypeDecorateRSLAstVisitor typeVisitor2 = new
                TM_TypeDecorateRSLAstVisitor(map, recordMapType,
```

```
            recordMapId, functionMapType, functionResultType,
            parentMap, ooExternalMethodList, true);
        rslast.accept(typeVisitor2);

        //System.out.println("map: " + map);
    }

    public void translate(String filename, boolean writeFiles) {
        FileReader fr = null;
        try {
            fr = new FileReader(filename + ".rsl");
        }
        catch(FileNotFoundException e) {
            System.out.println("File " + filename + ".rsl not
                found!");
            System.exit(1);
        }
        try {
            System.out.println("Starting: " + filename);
            //System.out.println("Starting parsing");
            RSLLexer lexer = new RSLLexer(fr);
            RSLParser parser = new RSLParser(lexer);
            TM_RSLAst rslast = parser.rslast();
            if(rslast == null) {
                System.out.println("Tree not created!");
                System.exit(1);
            }
            //System.out.println("RSLAST: " + rslast);
            //System.out.println("Ending parsing\n");
            //System.exit(1);

            //System.out.println("Starting setting parent of
                expressions");
            //TM_ParentRSLAstVisitor parentVisitor = new
                TM_ParentRSLAstVisitor();
            //rslast.accept(parentVisitor);
            //System.out.println("Ending setting parent of
                expressions\n");


            //System.out.println("Starting type decorating
                expressions");
            //typeDecorate(rslast);
            //System.out.println("Ending type decorating
                expressions\n");

            //System.out.println("Starting RSL");
            if((new Boolean(getProperty("printRSL"))).
                booleanValue()) {
```

```
                TM_StringRSLAstVisitor rslAstVisitor = new
                    TM_StringRSLAstVisitor();
                rslast.accept(rslAstVisitor);
                System.out.println("RSL:\n" + rslAstVisitor.
                    getResult());
            }
            //System.out.println("Ending RSL\n");

            //System.out.println("Starting translation");
            TM_JavaAst javaast = Translator_Module2.
                rslAst2JavaAst(rslast, opd, idl, ext, visitorId,
                ignoreList, writeExtensionFiles);
            //System.out.println("Ending translation\n");

            //System.out.println("JAVAAST: " + javaast);

            //System.out.println("Starting Java");
            TM_StringJavaAstVisitor javaAstVisitor = new
                TM_StringJavaAstVisitor(
                   (new Boolean(getProperty("printJava"))).
                       booleanValue(),
                   writeFiles,
                   getProperty("writeFilesDir")
            );
            javaast.accept(javaAstVisitor);
            if((new Boolean(getProperty("printJava"))).
                booleanValue()) {
                System.out.println("Java:\n" + javaAstVisitor.
                    getResult());
            }
            //System.out.println("Ending Java");
            System.out.println("Ending: " + filename);
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {

        instance = null;
        if(args.length == 1) {
            instance = new Runner2();
        }
        else if(args.length == 2) {
            instance = new Runner2(args[1]);
        }
        else {
            System.out.println("USAGE: translator.Runner2
```

```
                RSL_FILE_WITHOUT_EXTENSION [properties-file]");
            System.exit(1);
        }
        instance.translate(args[0], true);
    }
}
```

F.2 translator.lib

F.2.1 TM_JavaAstVisitor.java

```
package translator.lib;

import translator.javaast2.*;

public abstract class TM_JavaAstVisitor {

  public void visitTM_JavaAst(TM_JavaAst javaAst) {
    for (TM_CompilationUnit cu : javaAst.compilationUnitList().
        getList()) {
      cu.accept(this);
    }
  }

  public void visitTM_CompilationUnit(TM_CompilationUnit
      compilationUnit) {
    compilationUnit.optionalPackageDeclaration().accept(this);
    for (TM_ImportDeclaration id :
         compilationUnit.importDeclarationList().getList()) {
      id.accept(this);
    }
    for (TM_TypeDeclaration td : compilationUnit.
        typeDeclarationList().getList()) {
      td.accept(this);
    }
  }

  public void visitTM_ImportDeclaration(TM_ImportDeclaration
      importDeclaration) {
    importDeclaration.name().accept(this);
  }

  public void visitTM_PackageDeclaration(TM_PackageDeclaration
                                         packageDeclaration) {
    packageDeclaration.name().accept(this);
  }

  public void visitTM_NoPackageDeclaration(
```

```
    TM_NoPackageDeclaration
                                                noPackageDeclaration)
                                                    {}

  public void visitTM_ClassDeclaration(TM_ClassDeclaration
      classDeclaration) {
    for (TM_Modifier modifier : classDeclaration.modifierList().
        getList()) {
      modifier.accept(this);
    }
    classDeclaration.name().accept(this);
    if (classDeclaration.extendName() instanceof
        Make_TM_OptionalSimpleName) {
      classDeclaration.extendName().accept(this);
    }
    if (classDeclaration.implementList().len() > 0) {
      for (TM_SimpleName name : classDeclaration.implementList()
          .getList()) {
        name.accept(this);
      }
    }
    for (TM_MethodDeclaration methodDeclaration :
        classDeclaration.methodDeclarationList().getList()) {
      methodDeclaration.accept(this);
    }
    for (TM_FieldDeclaration fieldDeclaration :
        classDeclaration.fieldDeclarationList().getList()) {
      fieldDeclaration.accept(this);
    }
    for (TM_TypeDeclaration typeDeclaration :
        classDeclaration.typeDeclarationList().getList()) {
      typeDeclaration.accept(this);
    }
  }

  public void visitTM_FieldDeclaration(TM_FieldDeclaration
      fieldDeclaration) {
    for (TM_Modifier modifier : fieldDeclaration.modifierList().
        getList()) {
      modifier.accept(this);
    }
    fieldDeclaration.getType().accept(this);
    fieldDeclaration.variableDeclarationFragment().accept(this);
  }

  public void visitTM_MethodDeclaration(TM_MethodDeclaration
      methodDeclaration) {
    for (TM_Modifier modifier : methodDeclaration.modifierList()
        .getList()) {
```

```
        modifier.accept(this);
      }
      methodDeclaration.returnType().accept(this);
      methodDeclaration.name().accept(this);
      for (TM_SingleVariableDeclaration singleVariableDeclaration
          :
           methodDeclaration.argumentList().getList()) {
        singleVariableDeclaration.accept(this);
      }
      methodDeclaration.block().accept(this);
    }

    public void visitTM_ConstructorDeclaration(
        TM_ConstructorDeclaration
                                                constructorDeclaration
                                                    ) {
      for (TM_Modifier modifier : constructorDeclaration.
          modifierList().getList()) {
        modifier.accept(this);
      }
      constructorDeclaration.name().accept(this);
      for (TM_SingleVariableDeclaration singleVariableDeclaration
          :
           constructorDeclaration.argumentList().getList()) {
        singleVariableDeclaration.accept(this);
      }
      constructorDeclaration.block().accept(this);
    }

    public void visitTM_SingleVariableDeclaration(
        TM_SingleVariableDeclaration
                                                  singleVariableDeclaration
                                                      ) {
      for (TM_Modifier modifier :
           singleVariableDeclaration.modifierList().getList()) {
        modifier.accept(this);
      }
      singleVariableDeclaration.getType().accept(this);
      singleVariableDeclaration.name().accept(this);
      if (singleVariableDeclaration.optionalInitialization()
          instanceof
           Make_TM_OptionalExpression) {
        singleVariableDeclaration.optionalInitialization().accept(
            this);
      }
    }

    public void visitMake_TM_OptionalBlock(Make_TM_OptionalBlock
        optionalBlock) {
```

```
    optionalBlock.block().accept(this);
  }

  public void visitTM_NoOptionalBlock(TM_NoOptionalBlock
      noOptionalBlock) {}

  public void visitTM_Block(TM_Block block) {
    for (TM_Statement statement : block.statementList().getList
        ()) {
      statement.accept(this);
    }
  }

  public void visitMake_TM_Statement(Make_TM_Statement statement
      ) {
    statement.statement().accept(this);
  }

  public void visitTM_NoOptionalStatement(TM_NoOptionalStatement
                                          noOptionalStatement)
                                              {}

  public void visitTM_ExpressionStatement(TM_ExpressionStatement
                                          expressionStatement) {
    expressionStatement.expression().accept(this);
  }

  public void visitTM_IfStatement(TM_IfStatement ifStatement) {
    ifStatement.condition().accept(this);
    ifStatement.ifBlock().accept(this);
    if (ifStatement.elseBlock() instanceof Make_TM_OptionalBlock
        ) {
      ifStatement.elseBlock().accept(this);
    }
  }

  public void visitTM_ReturnStatement(TM_ReturnStatement
      returnStatement) {
    returnStatement.expressionReturnStatement().accept(this);
  }

  public void visitMake_TM_VariableDeclarationStatement(
      Make_TM_VariableDeclarationStatement
          makeVariableDeclarationStatement) {
    makeVariableDeclarationStatement.
        variableDeclarationStatement().accept(this);
  }

  public void visitTM_VariableDeclarationStatement(
```

```
        TM_VariableDeclarationStatement
            variableDeclarationStatement) {
      for (TM_Modifier modifier :
155         variableDeclarationStatement.modifierList().getList())
              {
        modifier.accept(this);
      }
      variableDeclarationStatement.getType().accept(this);
      variableDeclarationStatement.fragment().accept(this);
160   }

      public void visitTM_VariableDeclarationFragment(
          TM_VariableDeclarationFragment variableDeclarationFragment
              ) {
        variableDeclarationFragment.name().accept(this);
165     if (variableDeclarationFragment.optionalExpression()
            instanceof
            Make_TM_OptionalExpression) {
          variableDeclarationFragment.optionalExpression().accept(
              this);
        }
      }
170
      /*Expressions*/
      public void visitMake_TM_OptionalExpression(
          Make_TM_OptionalExpression
                                                  makeOptionalExpression
                                                      ) {
        makeOptionalExpression.expression().accept(this);
175   }

      public void visitTM_NoOptionalExpression(
          TM_NoOptionalExpression
                                                noOptionalExpression)
                                                    {}

180   public void visitMake_TM_JavaValueLiteral(
          Make_TM_JavaValueLiteral
                                                 makeValueLiteral) {
        makeValueLiteral.valueLiteral().accept(this);
      }

185   public void visitMake_TM_JavaValueLiteralInteger(
          Make_TM_JavaValueLiteralInteger makeValueLiteralInteger) {
        makeValueLiteralInteger.valueLiteralInteger().accept(this);
      }

190   public void visitTM_JavaValueLiteralInteger(
          TM_JavaValueLiteralInteger
```

```
                                                    valueLiteralInteger
                                                        ) {}

    public void visitMake_TM_JavaValueLiteralString(
        Make_TM_JavaValueLiteralString makeValueLiteralString) {
      makeValueLiteralString.valueLiteralString().accept(this);
    }

    public void visitTM_JavaValueLiteralString(
        TM_JavaValueLiteralString
                                                  valueLiteralString)
                                                      {}

    public void visitMake_TM_JavaValueLiteralChar(
        Make_TM_JavaValueLiteralChar
                                                     makeValueLiteralChar
                                                         ) {
      makeValueLiteralChar.valueLiteralChar().accept(this);
    }

    public void visitTM_JavaValueLiteralChar(
        TM_JavaValueLiteralChar
                                               valueLiteralChar) {}

    public void visitMake_TM_JavaValueLiteralDouble(
        Make_TM_JavaValueLiteralDouble makeValueLiteralDouble) {
      makeValueLiteralDouble.valueLiteralDouble().accept(this);
    }

    public void visitTM_JavaValueLiteralDouble(
        TM_JavaValueLiteralDouble
                                                  valueLiteralDouble)
                                                      {}

    public void visitMake_TM_JavaValueLiteralBool(
        Make_TM_JavaValueLiteralBool
                                                     makeValueLiteralBool
                                                         ) {
      makeValueLiteralBool.valueLiteralBool().accept(this);
    }

    public void visitTM_JavaValueLiteralBool(
        TM_JavaValueLiteralBool
                                               valueLiteralBool) {}

    public void visitTM_NullLiteral(TM_NullLiteral nullLiteral) {}

    public void visitTM_ArrayCreation(TM_ArrayCreation
        arrayCreation) {
```

```
    arrayCreation.getArrayType().accept(this);
    if (arrayCreation.getCount() instanceof
        Make_TM_OptionalExpression) {
      arrayCreation.getCount().accept(this);
    }
    if (arrayCreation.elementList().len() > 0) {
      for (TM_Expression expression : arrayCreation.elementList
          ().getList()) {
        expression.accept(this);
      }
    }
  }

  public void visitTM_InfixExpression(TM_InfixExpression
      infixExpression) {
    infixExpression.left().accept(this);
    infixExpression.op().accept(this);
    infixExpression.right().accept(this);
  }

  public void visitTM_PrefixExpression(TM_PrefixExpression
      prefixExpression) {
    prefixExpression.prefixOperator().accept(this);
    prefixExpression.prefixExpression().accept(this);
  }

  public void visitTM_MethodInvocation(TM_MethodInvocation
      methodInvocation) {
    methodInvocation.optionalExpression().accept(this);
    methodInvocation.name().accept(this);
    for (TM_Expression expression : methodInvocation.
        argumentList().getList()) {
      expression.accept(this);
    }
  }

  public void visitTM_ClassInstanceCreation(
      TM_ClassInstanceCreation
                                            classInstanceCreation
                                                ) {
    if (classInstanceCreation.
        optionalExpressionClassInstanceCreation() instanceof
         Make_TM_OptionalExpression) {
      classInstanceCreation.
          optionalExpressionClassInstanceCreation().accept(this);
    }
    classInstanceCreation.getType().accept(this);
    for (TM_Expression expression :
         classInstanceCreation.argumentListClassInstanceCreation
```

```
            ().getList()) {
        expression.accept(this);
      }
    }

    public void visitTM_AssignmentExpression(
        TM_AssignmentExpression
                                              assignmentExpression)
                                                  {
      assignmentExpression.lhs().accept(this);
      assignmentExpression.assignmentOp().accept(this);
      assignmentExpression.rhs().accept(this);
    }

    public void visitTM_ParenthesizedExpression(
        TM_ParenthesizedExpression
                                                parenthesizedExpression
                                                    ) {
      parenthesizedExpression.expression().accept(this);
    }

    public void visitTM_InstanceOfExpression(
        TM_InstanceOfExpression
                                              instanceOfExpression)
                                                  {
      instanceOfExpression.instanceOfExpression().accept(this);
      instanceOfExpression.instanceOfType().accept(this);
    }

    public void visitTM_ThisExpression(TM_ThisExpression
        thisExpression) {
      if (thisExpression.thisName() instanceof
          Make_TM_OptionalSimpleName) {
        thisExpression.thisName().accept(this);
      }
    }

    public void visitTM_CastExpression(TM_CastExpression
        castExpression) {
      castExpression.castType().accept(this);
      castExpression.castExpression().accept(this);
    }

    public void visitTM_FieldAccessExpression(
        TM_FieldAccessExpression
                                               fieldAccessExpression
                                                   ) {
      if (fieldAccessExpression.fieldExpression() instanceof
          Make_TM_OptionalExpression) {
```

```
        fieldAccessExpression.fieldExpression().accept(this);
305   }
      fieldAccessExpression.fieldName().accept(this);
    }

    public void visitTM_JAVA_NOT(TM_JAVA_NOT not) {}
310
    public void visitTM_JAVA_EQUALS(TM_JAVA_EQUALS equals) {}

    public void visitTM_JAVA_PLUS(TM_JAVA_PLUS plus) {}

315 public void visitTM_JAVA_STAR(TM_JAVA_STAR star) {}

    public void visitTM_JAVA_DIV(TM_JAVA_DIV div) {}

    public void visitTM_JAVA_ASSIGNMENT_OP_EQUAL(
        TM_JAVA_ASSIGNMENT_OP_EQUAL
320                                                 equal) {}

    /*NAMES*/
    public void visitMake_TM_JavaName(Make_TM_JavaName
        makeJavaName) {
      makeJavaName.name().accept(this);
325 }

    public void visitMake_TM_SimpleName(Make_TM_SimpleName
        makeSimpleName) {
      makeSimpleName.name().accept(this);
    }
330
    public void visitMake_TM_QualifiedName(Make_TM_QualifiedName
                                           makeQualifiedName) {
      makeQualifiedName.name().accept(this);
    }
335
    public void visitTM_QualifiedName(TM_QualifiedName
        qualifiedName) {
      qualifiedName.left().accept(this);
      qualifiedName.right().accept(this);
    }
340
    public void visitTM_SimpleName(TM_SimpleName simpleName) {}

    public void visitTM_NoOptionalSimpleName(
        TM_NoOptionalSimpleName
                                             noOptionalSimpleName)
                                                 {}
345
    public void visitMake_TM_OptionalSimpleName(
```

```
    Make_TM_OptionalSimpleName
                                                    optionalSimpleName
                                                        ) {
  optionalSimpleName.name().accept(this);
}

/*Types*/
public void visitMake_TM_ReferenceType(Make_TM_ReferenceType
                                                    makeReferenceType) {
  makeReferenceType.referenceType().accept(this);
}

public void visitMake_TM_ArrayType(Make_TM_ArrayType
    makeArrayType) {
  makeArrayType.arrayType().accept(this);
}

public void visitMake_TM_PrimitiveType(Make_TM_PrimitiveType
                                                    makePrimitiveType) {
  makePrimitiveType.primitiveType().accept(this);
}

public void visitTM_ReferenceType(TM_ReferenceType
    referenceType) {
  referenceType.name().accept(this);
  if (referenceType.optionalTypeArgument() instanceof
      Make_TM_OptionalReferenceType) {
    referenceType.optionalTypeArgument().accept(this);
  }
}

public void visitTM_ArrayType(TM_ArrayType arrayType) {
  arrayType.getType().accept(this);
}

public void visitMake_TM_OptionalReferenceType(
    Make_TM_OptionalReferenceType
                                                    optionalReferenceType
                                                        ) {
  optionalReferenceType.referenceType().accept(this);
}

public void visitTM_NoOptionalReferenceType(
    TM_NoOptionalReferenceType
                                                    noOptionalReferenceType
                                                        ) {}

public void visitTM_JAVA_INT(TM_JAVA_INT java_int) {}
```

```
  public void visitTM_JAVA_VOID(TM_JAVA_VOID java_void) {}

  public void visitTM_JAVA_DOUBLE(TM_JAVA_DOUBLE java_double) {}

  public void visitTM_JAVA_BOOL(TM_JAVA_BOOL java_bool) {}

  public void visitTM_JAVA_CHAR(TM_JAVA_CHAR java_char) {}

  /*Other*/
  public void visitTM_JAVA_PUBLIC(TM_JAVA_PUBLIC java_public) {}

  public void visitTM_JAVA_STATIC(TM_JAVA_STATIC java_static) {}

  public void visitTM_JAVA_ABSTRACT(TM_JAVA_ABSTRACT
      java_abstract) {}

  public void visitTM_JAVA_PRIVATE(TM_JAVA_PRIVATE java_private)
      {}

}
```

F.2.2 TM_ParentRSLAstVisitor.java

```
package translator.lib;

import translator.rslast2.*;

public class TM_ParentRSLAstVisitor
    extends TM_RSLAstVisitor {

  public TM_ParentRSLAstVisitor() {}

  public void visitTM_RSLAst(TM_RSLAst rslast) {
    rslast.libmodule().accept(this);
    rslast.libmodule().setParent(rslast);
  }

  public void visitTM_LibModule(TM_LibModule module) {
    module.schemedef().accept(this);
    module.schemedef().setParent(module);
  }

  public void visitTM_SchemeDef(TM_SchemeDef scheme) {
    scheme.id().accept(this);
    scheme.class_expr().accept(this);
    scheme.id().setParent(scheme);
    scheme.class_expr().setParent(scheme);
  }
```

```
  public void visitTM_BasicClassExpr(TM_BasicClassExpr
      basicClassExpr) {
    for (TM_Decl decl : basicClassExpr.declaration_list().
        getList()) {
      decl.accept(this);
      decl.setParent(basicClassExpr);
    }
  }

  /*Type Declarations */
  public void visitTM_TypeDecl(TM_TypeDecl typeDecl) {
    for (TM_TypeDef typeDef : typeDecl.type_def_list().getList()
        ) {
      typeDef.accept(this);
      typeDef.setParent(typeDecl);
    }
  }

  public void visitTM_SortDef(TM_SortDef sortDef) {
    sortDef.sd_id().accept(this);
  }

  public void visitTM_VariantDef(TM_VariantDef variantDef) {
    variantDef.id().accept(this);
    for (TM_Variant variant : variantDef.variant_list().getList
        ()) {
      variant.setParent(variantDef);
      variant.accept(this);
    }
  }

  public void visitTM_ShortRecordDef(TM_ShortRecordDef
      shortRecordDef) {
    shortRecordDef.srd_id().accept(this);
    for (TM_ComponentKind componentKind :
         shortRecordDef.component_kind_string().getList()) {
      componentKind.accept(this);
      componentKind.setParent(shortRecordDef);
    }
  }

  public void visitMake_TM_Constructor(Make_TM_Constructor
      constructor) {
    constructor.constructor().accept(this);
    constructor.constructor().setParent(constructor.getParent())
        ;
  }

  public void visitTM_Constructor(TM_Constructor constructor) {
```

```
    constructor.id().accept(this);
  }

  public void visitTM_Destructor(TM_Destructor destructor) {
    destructor.id().accept(this);
  }

  public void visitTM_RecordVariant(TM_RecordVariant
      recordVariant) {
    recordVariant.record_constructor().accept(this);
    recordVariant.record_constructor().setParent(recordVariant);
    for (TM_ComponentKind componentKind :
        recordVariant.component_kind_list().getList()) {
      componentKind.accept(this);
      componentKind.setParent(recordVariant);
    }
  }

  public void visitTM_ComponentKind(TM_ComponentKind
      componentKind) {
    componentKind.type_expr().accept(this);
    componentKind.optional_destructor().accept(this);
    componentKind.optional_reconstructor().accept(this);
  }

  /*Value Declarations*/
  public void visitTM_ValueDecl(TM_ValueDecl valueDecl) {
    for (TM_ValueDef valueDef : valueDecl.value_def_list().
        getList()) {
      valueDef.accept(this);
      valueDef.setParent(valueDecl);
    }
  }

  public void visitTM_ExplicitFunctionDef(TM_ExplicitFunctionDef
                                          explicitFunctionDef) {
    explicitFunctionDef.single_typing().accept(this);
    explicitFunctionDef.formal_function_application().accept(
        this);
    explicitFunctionDef.value_expr().accept(this);
    explicitFunctionDef.single_typing().setParent(
        explicitFunctionDef);
    explicitFunctionDef.formal_function_application().setParent(
        explicitFunctionDef);
    explicitFunctionDef.value_expr().setParent(
        explicitFunctionDef);
  }

  public void visitTM_SingleTyping(TM_SingleTyping singleTyping)
```

```
        {
      singleTyping.binding().accept(this);
      singleTyping.type_expr().accept(this);
      singleTyping.binding().setParent(singleTyping);
      singleTyping.type_expr().setParent(singleTyping);
    }

    public void visitTM_IdApplication(TM_IdApplication
        idApplication) {
      idApplication.id().accept(this);
      idApplication.id().setParent(idApplication);
      for (TM_FormalFunctionParameter parameter :
          idApplication.formal_function_parameter_list().getList
              ()) {
        parameter.accept(this);
        parameter.setParent(idApplication);
      }
    }

    public void visitTM_FormalFunctionParameter(
        TM_FormalFunctionParameter
                                                formalFunctionParameter
                                                    ) {
      for (TM_Binding binding : formalFunctionParameter.
          binding_list().getList()) {
        binding.accept(this);
        binding.setParent(formalFunctionParameter);
      }
    }

    /*Type Expression*/
    public void visitTM_TypeExprList(TM_TypeExprList typeExprList)
        {
      typeExprList.type_expr_list().accept(this);
      typeExprList.type_expr_list().setParent(typeExprList);

    }

    public void visitTM_FiniteListTypeExpr(TM_FiniteListTypeExpr
                                           finiteListTypeExpr) {
      finiteListTypeExpr.type_expr().accept(this);
      finiteListTypeExpr.type_expr().setParent(finiteListTypeExpr)
          ;
    }

    public void visitTM_FunctionTypeExpr(TM_FunctionTypeExpr
        functionTypeExpr) {
      functionTypeExpr.type_expr_argument().accept(this);
      functionTypeExpr.function_arrow().accept(this);
```

```
    functionTypeExpr.type_expr_result().accept(this);

    functionTypeExpr.type_expr_argument().setParent(
        functionTypeExpr);
    functionTypeExpr.function_arrow().setParent(functionTypeExpr
        );
    functionTypeExpr.type_expr_result().setParent(
        functionTypeExpr);
  }

  public void visitTM_TOTAL_FUNCTION_ARROW(
      TM_TOTAL_FUNCTION_ARROW
                                           totalFunctionArrow)
                                               {}

  public void visitTM_TypeLiteral(TM_TypeLiteral typeLiteral) {
    typeLiteral.type_literal().accept(this);
    typeLiteral.type_literal().setParent(typeLiteral);
  }

  public void visitTM_RSL_UNIT(TM_RSL_UNIT rsl_unit) {}

  public void visitTM_RSL_INT(TM_RSL_INT rsl_int) {}

  public void visitTM_RSL_NAT(TM_RSL_NAT rsl_nat) {}

  public void visitTM_RSL_REAL(TM_RSL_REAL rsl_real) {}

  public void visitTM_RSL_BOOL(TM_RSL_BOOL rsl_bool) {}

  public void visitTM_RSL_CHAR(TM_RSL_CHAR rsl_char) {}

  public void visitTM_RSL_TEXT(TM_RSL_TEXT rsl_text) {}

  /*Value Expression*/
  public void visitTM_ApplicationExpr(TM_ApplicationExpr
      applicationExpr) {
    applicationExpr.value_expr().accept(this);
    applicationExpr.setParent(applicationExpr);
    for (TM_ValueExpr ve : applicationExpr.value_expr_list().
        getList()) {
      ve.accept(this);
      ve.setParent(applicationExpr);
    }
  }

  public void visitMake_TM_ListExpr(Make_TM_ListExpr
      make_ListExpr) {
    make_ListExpr.list_expr().accept(this);
```

```
    make_ListExpr.list_expr().setParent(make_ListExpr);
  }

  public void visitTM_EnumeratedListExpr(TM_EnumeratedListExpr
                                         enumeratedListExpr) {
    for (TM_ValueExpr ve : enumeratedListExpr.value_expr_list().
        getList()) {
      ve.accept(this);
      ve.setParent(enumeratedListExpr);
    }
  }

  public void visitMake_TM_IfExpr(Make_TM_IfExpr makeIfExpr) {
    makeIfExpr.if_expr().accept(this);
    makeIfExpr.if_expr().setParent(makeIfExpr);
  }

  public void visitTM_IfExpr(TM_IfExpr ifExpr) {
    ifExpr.condition().accept(this);
    ifExpr.if_case().accept(this);
    for (TM_Elsif elsif : ifExpr.elsif_list().getList()) {
      elsif.accept(this);
    }
    ifExpr.else_case().accept(this);
    ifExpr.condition().setParent(ifExpr);
    ifExpr.if_case().setParent(ifExpr);
    for (TM_Elsif elsif : ifExpr.elsif_list().getList()) {
      elsif.accept(this);
      elsif.setParent(ifExpr);
    }
    ifExpr.else_case().setParent(ifExpr);

  }

  public void visitTM_Elsif(TM_Elsif elsif) {
    elsif.condition().accept(this);
    elsif.elsif_case().accept(this);
    elsif.condition().setParent(elsif);
    elsif.elsif_case().setParent(elsif);
  }

  public void visitTM_CaseExpr(TM_CaseExpr caseExpr) {
    caseExpr.condition().accept(this);
    caseExpr.condition().setParent(caseExpr);
    for (TM_CaseBranch cb : caseExpr.case_branch_list().getList
        ()) {
      cb.accept(this);
      cb.setParent(caseExpr);
    }
```

```
240   }

      public void visitTM_CaseBranch(TM_CaseBranch caseBranch) {
        caseBranch.pattern().accept(this);
        caseBranch.value_expr().accept(this);
245     caseBranch.pattern().setParent(caseBranch);
        caseBranch.value_expr().setParent(caseBranch);
      }

      public void visitTM_ValueLiteralPattern(TM_ValueLiteralPattern
          pattern) {
250     pattern.value_literal().accept(this);
        pattern.value_literal().setParent(pattern);
      }

      public void visitTM_RecordPattern(TM_RecordPattern pattern) {
255     pattern.value_or_variable_name().accept(this);
        pattern.value_or_variable_name().setParent(pattern);
        for (TM_Pattern ip : pattern.inner_pattern_list().getList())
            {
          ip.accept(this);
          ip.setParent(pattern);
260     }
      }

      public void visitTM_NamePattern(TM_NamePattern pattern) {
        pattern.id().accept(this);
265     pattern.id().setParent(pattern);
      }

      public void visitTM_WildcardPattern(TM_WildcardPattern pattern
          ) {}

270   public void visitTM_ValueInfixExpr(TM_ValueInfixExpr
          valueInfixExpr) {
        valueInfixExpr.left().accept(this);
        valueInfixExpr.op().accept(this);
        valueInfixExpr.right().accept(this);
        valueInfixExpr.left().setParent(valueInfixExpr);
275     valueInfixExpr.op().setParent(valueInfixExpr);
        valueInfixExpr.right().setParent(valueInfixExpr);
      }

      public void visitTM_ValuePrefixExpr(TM_ValuePrefixExpr
          valuePrefixExpr) {
280     valuePrefixExpr.op().accept(this);
        valuePrefixExpr.operand().accept(this);
        valuePrefixExpr.op().setParent(valuePrefixExpr);
        valuePrefixExpr.operand().setParent(valuePrefixExpr);
```

```
  }

  public void visitTM_ParenthesizedExpr(TM_ParenthesizedExpr
      parenthesizedExpr) {
    parenthesizedExpr.parenthesized_expr().accept(this);
    parenthesizedExpr.parenthesized_expr().setParent(
        parenthesizedExpr);
  }

  public void visitTM_RSL_EQUAL(TM_RSL_EQUAL rsl_equal) {}

  public void visitTM_RSL_PLUS(TM_RSL_PLUS rsl_plus) {}

  public void visitTM_RSL_STAR(TM_RSL_STAR rsl_star) {}

  public void visitTM_RSL_HAT(TM_RSL_HAT rsl_hat) {}

  public void visitTM_RSL_HD(TM_RSL_HD rsl_hd) {}

  public void visitTM_RSL_TL(TM_RSL_TL rsl_tl) {}

  public void visitMake_TM_ValueOrVariableName(
      Make_TM_ValueOrVariableName
                                                valueOrVariableName
                                                    ) {
    valueOrVariableName.value_or_variable_name().accept(this);
    valueOrVariableName.value_or_variable_name().setParent(
        valueOrVariableName);
  }

  public void visitTM_ValueOrVariableName(TM_ValueOrVariableName
                                            valueOrVariableName) {
    valueOrVariableName.id().accept(this);
    valueOrVariableName.id().setParent(valueOrVariableName);
  }

  public void visitTM_ValueLiteralInteger(TM_ValueLiteralInteger
                                            valueLiteralInteger)
                                                {}

  /*Common*/
  public void visitTM_Binding(TM_Binding binding) {}

  public void visitTM_Id(TM_Id id) {}
}
```

F.2.3 TM_RSLAstVisitor.java

```
package translator.lib;
```

```
import translator.rslast2.*;

public abstract class TM_RSLAstVisitor {

  public void visitTM_RSLAst(TM_RSLAst rslast) {
    rslast.libmodule().accept(this);
  }

  public void visitTM_LibModule(TM_LibModule module) {
    for (TM_Id id : module.context_list().getList()) {
      id.accept(this);
    }
    module.schemedef().accept(this);
  }

  public void visitTM_SchemeDef(TM_SchemeDef scheme) {
    scheme.id().accept(this);
    scheme.class_expr().accept(this);
  }

  public void visitTM_ExtendingClassExpr(TM_ExtendingClassExpr
                                         extendingClassExpr) {
    extendingClassExpr.base_class().accept(this);
    extendingClassExpr.extension_class().accept(this);
  }

  public void visitTM_SchemeInstantiation(TM_SchemeInstantiation
                                          schemeInstantiation) {
    schemeInstantiation.id().accept(this);
  }

  public void visitTM_BasicClassExpr(TM_BasicClassExpr
      basicClassExpr) {
    for (TM_Decl decl : basicClassExpr.declaration_list().
        getList()) {
      decl.accept(this);
    }
  }

  /*Type Declarations */
  public void visitTM_TypeDecl(TM_TypeDecl typeDecl) {
    for (TM_TypeDef typeDef : typeDecl.type_def_list().getList()
        ) {
      typeDef.accept(this);
    }
  }

  public void visitTM_SortDef(TM_SortDef sortDef) {
```

```
    sortDef.sd_id().accept(this);
  }

  public void visitTM_ShortRecordDef(TM_ShortRecordDef
      shortRecordDef) {
    shortRecordDef.srd_id().accept(this);
    for (TM_ComponentKind componentKind :
        shortRecordDef.component_kind_string().getList()) {
      componentKind.accept(this);
    }
  }

  public void visitTM_VariantDef(TM_VariantDef variantDef) {
    variantDef.id().accept(this);
    for (TM_Variant variant : variantDef.variant_list().getList
        ()) {
      variant.accept(this);
    }
  }

  public void visitTM_RecordVariant(TM_RecordVariant
      recordVariant) {
    recordVariant.record_constructor().accept(this);
    for (TM_ComponentKind componentKind :
        recordVariant.component_kind_list().getList()) {
      componentKind.accept(this);
    }
  }

  public void visitTM_ComponentKind(TM_ComponentKind
      componentKind) {
    if (componentKind.optional_destructor() instanceof
        TM_Destructor) {
      componentKind.optional_destructor().accept(this);
    }
    componentKind.type_expr().accept(this);
    if (componentKind.optional_reconstructor() instanceof
        TM_Reconstructor) {
      componentKind.optional_reconstructor().accept(this);
    }
  }

  public void visitMake_TM_Constructor(Make_TM_Constructor
      constructor) {
    constructor.constructor().accept(this);
  }

  public void visitTM_Constructor(TM_Constructor constructor) {
    constructor.id().accept(this);
```

```
90    }

      public void visitTM_Destructor(TM_Destructor destructor) {
        destructor.id().accept(this);
      }
95
      public void visitTM_NoDestructor(TM_NoDestructor noDestructor)
          {}

      public void visitTM_Reconstructor(TM_Reconstructor
          reconstructor) {
        reconstructor.id().accept(this);
100   }

      public void visitTM_NoReconstructor(TM_NoReconstructor
          noReconstructor) {}

      /*Value Declarations*/
105   public void visitTM_ValueDecl(TM_ValueDecl valueDecl) {
        for (TM_ValueDef valueDef : valueDecl.value_def_list().
            getList()) {
          valueDef.accept(this);
        }
      }
110
      public void visitTM_ExplicitFunctionDef(TM_ExplicitFunctionDef
                                              explicitFunctionDef) {
        explicitFunctionDef.single_typing().accept(this);
        explicitFunctionDef.formal_function_application().accept(
            this);
115     explicitFunctionDef.value_expr().accept(this);
      }

      public void visitTM_SingleTyping(TM_SingleTyping singleTyping)
          {
        singleTyping.binding().accept(this);
120     singleTyping.type_expr().accept(this);
      }

      public void visitTM_IdApplication(TM_IdApplication
          idApplication) {
        idApplication.id().accept(this);
125     for (TM_FormalFunctionParameter parameter :
              idApplication.formal_function_parameter_list().getList
                  ()) {
          parameter.accept(this);
        }
      }
130
```

```
  public void visitTM_FormalFunctionParameter(
      TM_FormalFunctionParameter
                                                formalFunctionParameter
                                                  ) {
    for (TM_Binding binding : formalFunctionParameter.
        binding_list().getList()) {
      binding.accept(this);
    }
  }

  /*Test Declarations*/
  public void visitTM_TestDecl(TM_TestDecl testDecl) {
    for (TM_TestDef testDef : testDecl.test_def_list().getList()
        ) {
      testDef.accept(this);
    }
  }

  public void visitTM_TestCase(TM_TestCase testCase) {
    testCase.id().accept(this);
    testCase.value_expr().accept(this);
  }

  /*Type Expression*/
  public void visitTM_TypeExprList(TM_TypeExprList typeExprList)
      {
    typeExprList.type_expr_list().accept(this);
  }

  public void visitTM_FiniteListTypeExpr(TM_FiniteListTypeExpr
                                           finiteListTypeExpr) {
    finiteListTypeExpr.type_expr().accept(this);
  }

  public void visitTM_FunctionTypeExpr(TM_FunctionTypeExpr
      functionTypeExpr) {
    functionTypeExpr.type_expr_argument().accept(this);
    functionTypeExpr.function_arrow().accept(this);
    functionTypeExpr.type_expr_result().accept(this);
  }

  public void visitTM_TOTAL_FUNCTION_ARROW(
      TM_TOTAL_FUNCTION_ARROW
                                             totalFunctionArrow)
                                               {}

  public void visitTM_TypeExprProduct(TM_TypeExprProduct
      typeExprProduct) {
    for (TM_TypeExpr te : typeExprProduct.component_list().
```

```
      getList()) {
    te.accept(this);
  }
}

public void visitTM_TypeLiteral(TM_TypeLiteral typeLiteral) {
  typeLiteral.type_literal().accept(this);
}

public void visitTM_TypeName(TM_TypeName typeName) {
  typeName.id().accept(this);
}

public void visitTM_RSL_UNIT(TM_RSL_UNIT rsl_unit) {}

public void visitTM_RSL_INT(TM_RSL_INT rsl_int) {}

public void visitTM_RSL_NAT(TM_RSL_NAT rsl_nat) {}

public void visitTM_RSL_REAL(TM_RSL_REAL rsl_real) {}

public void visitTM_RSL_BOOL(TM_RSL_BOOL rsl_bool) {}

public void visitTM_RSL_CHAR(TM_RSL_CHAR rsl_char) {}

public void visitTM_RSL_TEXT(TM_RSL_TEXT rsl_text) {}

/*Value Expression*/
public void visitTM_ApplicationExpr(TM_ApplicationExpr
    applicationExpr) {
  applicationExpr.value_expr().accept(this);
  for (TM_ValueExpr ve : applicationExpr.value_expr_list().
      getList()) {
    ve.accept(this);

  }
}

public void visitMake_TM_ListExpr(Make_TM_ListExpr
    make_ListExpr) {
  make_ListExpr.list_expr().accept(this);
}

public void visitTM_EnumeratedListExpr(TM_EnumeratedListExpr
                                       enumeratedListExpr) {
  for (TM_ValueExpr ve : enumeratedListExpr.value_expr_list().
      getList()) {
    ve.accept(this);
  }
```

```
215   }

      public void visitMake_TM_IfExpr(Make_TM_IfExpr makeIfExpr) {
        makeIfExpr.if_expr().accept(this);
      }
220
      public void visitTM_IfExpr(TM_IfExpr ifExpr) {
        ifExpr.condition().accept(this);
        ifExpr.if_case().accept(this);
        for (TM_Elsif elsif : ifExpr.elsif_list().getList()) {
225       elsif.accept(this);
        }
        ifExpr.else_case().accept(this);
      }

230   public void visitTM_Elsif(TM_Elsif elsif) {
        elsif.condition().accept(this);
        elsif.elsif_case().accept(this);
      }

235   public void visitTM_CaseExpr(TM_CaseExpr caseExpr) {
        caseExpr.condition().accept(this);
        for (TM_CaseBranch cb : caseExpr.case_branch_list().getList
            ()) {
          cb.accept(this);
        }
240   }

      public void visitTM_CaseBranch(TM_CaseBranch caseBranch) {
        caseBranch.pattern().accept(this);
        caseBranch.value_expr().accept(this);
245   }

      public void visitTM_ValueLiteralPattern(TM_ValueLiteralPattern
          pattern) {
        pattern.value_literal().accept(this);
      }
250
      public void visitTM_RecordPattern(TM_RecordPattern pattern) {
        pattern.value_or_variable_name().accept(this);
      }

255   public void visitTM_NamePattern(TM_NamePattern pattern) {
        pattern.id().accept(this);
      }

      public void visitTM_WildcardPattern(TM_WildcardPattern pattern
          ) {}
260
```

```
  public void visitTM_ValueInfixExpr(TM_ValueInfixExpr
      valueInfixExpr) {
    valueInfixExpr.left().accept(this);
    valueInfixExpr.op().accept(this);
    valueInfixExpr.right().accept(this);
  }

  public void visitTM_ValuePrefixExpr(TM_ValuePrefixExpr
      valuePrefixExpr) {
    valuePrefixExpr.op().accept(this);
    valuePrefixExpr.operand().accept(this);
  }

  public void visitTM_ParenthesizedExpr(TM_ParenthesizedExpr
      parenthesizedExpr) {
    parenthesizedExpr.parenthesized_expr().accept(this);
  }

  public void visitTM_RSL_EQUAL(TM_RSL_EQUAL rsl_equal) {}

  public void visitTM_RSL_PLUS(TM_RSL_PLUS rsl_plus) {}

  public void visitTM_RSL_STAR(TM_RSL_STAR rsl_star) {}

  public void visitTM_RSL_HAT(TM_RSL_HAT rsl_hat) {}

  public void visitTM_RSL_SLASH(TM_RSL_SLASH rsl_slash) {}

  public void visitTM_RSL_HD(TM_RSL_HD rsl_hd) {}

  public void visitTM_RSL_TL(TM_RSL_TL rsl_tl) {}

  public void visitMake_TM_ValueOrVariableName(
      Make_TM_ValueOrVariableName
                                                valueOrVariableName
                                                    ) {
    valueOrVariableName.value_or_variable_name().accept(this);
  }

  public void visitTM_ValueOrVariableName(TM_ValueOrVariableName
                                            valueOrVariableName) {
    valueOrVariableName.id().accept(this);
  }

  public void visitMake_TM_ValueLiteral(Make_TM_ValueLiteral
      valueLiteral) {
    valueLiteral.value_literal().accept(this);
  }
```

```
  public void visitTM_ValueLiteralInteger(TM_ValueLiteralInteger
305                                       valueLiteralInteger)
                                            {}

  public void visitTM_ValueLiteralReal(TM_ValueLiteralReal
      valueLiteralReal) {}

  public void visitTM_ValueLiteralBool(TM_ValueLiteralBool
      valueLiteralBool) {}
310
  public void visitTM_ValueLiteralChar(TM_ValueLiteralChar
      valueLiteralChar) {}

  public void visitTM_ValueLiteralText(TM_ValueLiteralText
      valueLiteralText) {}

315 /*Common*/
  public void visitTM_Binding(TM_Binding binding) {}

  public void visitMake_TM_Id(Make_TM_Id id) {
    id.id().accept(this);
320 }

  public void visitTM_NoOptionalId(TM_NoOptionalId noId) {}

  public void visitTM_Id(TM_Id id) {}
325 }
```

F.2.4 TM_RSLElement.java

```
package translator.lib;

import translator.rslast2.*;
import translator.*;
5
public abstract class TM_RSLElement {
  private TM_RSLElement parent;
  private TM_TypeEvaluator typeEvaluator;

10 public TM_RSLElement getParent() {
    return parent;
  }

  public void setParent(TM_RSLElement parent) {
15   this.parent = parent;
  }

  public TM_TypeEvaluator getTypeEvaluatorForTypeEvaluation() {
    return typeEvaluator;
```

```
20    }

      public void setTypeEvaluatorForTypeEvaluation(TM_TypeEvaluator
          typeEvaluator) {
        this.typeEvaluator = typeEvaluator;
      }
25
      public abstract void accept(TM_RSLAstVisitor visitor);
    }
```

F.2.5 TM_StringJavaAstVisitor.java

```
    package translator.lib;

    import java.util.*;
    import java.io.*;
5   import java.text.*;

    import translator.javaast2.*;

    public class TM_StringJavaAstVisitor
10      extends TM_JavaAstVisitor {
      private StringBuffer result;
      private StringBuffer complete;
      private String dir;
      private boolean writeFiles;
15    private boolean printJava;

      public TM_StringJavaAstVisitor(boolean printJava, boolean
          writeFiles,
                                     String dir) {
        this.printJava = printJava;
20      this.writeFiles = writeFiles;
        this.dir = dir;
        this.result = new StringBuffer();
        this.complete = new StringBuffer();
      }
25
      public String getResult() {
        if (writeFiles) {
          complete.append(result.toString());
          return complete.toString();
30      }
        else {
          return result.toString();
        }
      }
35
      public void visitTM_JavaAst(TM_JavaAst javaAst) {
```

```
    for (TM_CompilationUnit cu : javaAst.compilationUnitList().
        getList()) {
      cu.accept(this);
    }
  }

  public void visitTM_CompilationUnit(TM_CompilationUnit
      compilationUnit) {
    if (writeFiles) {
      complete.append(result.toString());
      result = new StringBuffer();
    }
    compilationUnit.optionalPackageDeclaration().accept(this);
    for (TM_ImportDeclaration id :
        compilationUnit.importDeclarationList().getList()) {
      id.accept(this);
      result.append("\n");
    }
    result.append("\n");
    for (TM_TypeDeclaration td : compilationUnit.
        typeDeclarationList().getList()) {
      td.accept(this);
      result.append("\n");
    }

    if (writeFiles) {
      for (TM_TypeDeclaration td :
          compilationUnit.typeDeclarationList().getList()) {
        if (td instanceof TM_ClassDeclaration) {
          TM_ClassDeclaration cd = (TM_ClassDeclaration) td;
          if (cd.modifierList().len() > 0 &&
              cd.modifierList().hd() instanceof TM_JAVA_PUBLIC)
                  {
            try {
              FileWriter fw = new FileWriter(dir + "/" + cd.name
                  ().text() +
                                             ".java", false);
              fw.write("//" +
                       DateFormat.getDateTimeInstance(DateFormat
                           .LONG,
                   DateFormat.LONG, Locale.UK).format(new Date())
                       + "\n");
              fw.write(this.result.toString());
              fw.flush();
              fw.close();
            }
            catch (IOException ioe) {
              ioe.printStackTrace();
            }
```

```
            }
          }
        }
      }
    }

    public void visitTM_ImportDeclaration(TM_ImportDeclaration
        importDeclaration) {
      result.append("import ");
      importDeclaration.name().accept(this);
      result.append(";");
    }

    public void visitTM_PackageDeclaration(TM_PackageDeclaration
                                           packageDeclaration) {
      result.append("package ");
      packageDeclaration.name().accept(this);
      result.append(";\n");
    }

    public void visitTM_NoPackageDeclaration(
        TM_NoPackageDeclaration
                                             noPackageDeclaration)
                                               {
      result.append("//NO PACKAGE DECLARATION\n");
    }

    public void visitTM_ClassDeclaration(TM_ClassDeclaration
        classDeclaration) {
      for (TM_Modifier modifier : classDeclaration.modifierList().
          getList()) {
        modifier.accept(this);
        result.append(" ");
      }
      result.append("class");
      result.append(" ");
      classDeclaration.name().accept(this);
      if (classDeclaration.extendName() instanceof
          Make_TM_OptionalSimpleName) {
        result.append(" ");
        result.append("extends");
        result.append(" ");
        classDeclaration.extendName().accept(this);
      }
      result.append(" ");
      if (classDeclaration.implementList().len() > 0) {
        result.append("implements");
        result.append(" ");
        for (TM_SimpleName name : classDeclaration.implementList()
```

```
            .getList()) {
          name.accept(this);
          result.append(" ");
        }
      }
      result.append("{\n");
      for (TM_FieldDeclaration fieldDeclaration :
          classDeclaration.fieldDeclarationList().getList()) {
        fieldDeclaration.accept(this);
        result.append("\n");
      }
      for (TM_ConstructorDeclaration constructorDeclaration :
          classDeclaration.constructorDeclarationList().getList()
              ) {
        constructorDeclaration.accept(this);
        result.append("\n");
      }
      for (TM_MethodDeclaration methodDeclaration :
          classDeclaration.methodDeclarationList().getList()) {
        methodDeclaration.accept(this);
        result.append("\n");
      }
      for (TM_TypeDeclaration typeDeclaration :
          classDeclaration.typeDeclarationList().getList()) {
        typeDeclaration.accept(this);
        result.append("\n");
      }
      result.append("}");
    }

    public void visitTM_FieldDeclaration(TM_FieldDeclaration
        fieldDeclaration) {
      result.append("//FIELD DECLARATION\n");
      for (TM_Modifier modifier : fieldDeclaration.modifierList().
          getList()) {
        modifier.accept(this);
        result.append(" ");
      }
      fieldDeclaration.getType().accept(this);
      result.append(" ");
      fieldDeclaration.variableDeclarationFragment().accept(this);
      result.append(";");
    }

    public void visitTM_MethodDeclaration(TM_MethodDeclaration
        methodDeclaration) {
      result.append("//METHOD DECLARATION\n");
      for (TM_Modifier modifier : methodDeclaration.modifierList()
          .getList()) {
```

```
165         modifier.accept(this);
            result.append(" ");
          }
          methodDeclaration.returnType().accept(this);
          result.append(" ");
170       methodDeclaration.name().accept(this);
          result.append("(");
          for (TM_SingleVariableDeclaration singleVariableDeclaration
              :
               methodDeclaration.argumentList().getList()) {
            singleVariableDeclaration.accept(this);
175         result.append(", ");
          }
          if (methodDeclaration.argumentList().len() > 0) {
            result.delete(result.length() - 2, result.length());
          }
180       result.append(")");
          result.append(" ");
          methodDeclaration.block().accept(this);
        }

185     public void visitTM_ConstructorDeclaration(
            TM_ConstructorDeclaration
                                                  constructorDeclaration
                                                     ) {
          for (TM_Modifier modifier : constructorDeclaration.
              modifierList().getList()) {
            modifier.accept(this);
            result.append(" ");
190       }
          constructorDeclaration.name().accept(this);
          result.append("(");
          for (TM_SingleVariableDeclaration singleVariableDeclaration
              :
               constructorDeclaration.argumentList().getList()) {
195         singleVariableDeclaration.accept(this);
            result.append(", ");
          }
          if (constructorDeclaration.argumentList().len() > 0) {
            result.delete(result.length() - 2, result.length());
200       }
          result.append(")");
          result.append(" ");
          constructorDeclaration.block().accept(this);
        }
205
        public void visitTM_SingleVariableDeclaration(
            TM_SingleVariableDeclaration
                                                    singleVariableDeclaration
```

```
                                                        ) {
    for (TM_Modifier modifier :
         singleVariableDeclaration.modifierList().getList()) {
      modifier.accept(this);
      result.append(" ");
    }
    singleVariableDeclaration.getType().accept(this);
    result.append(" ");
    singleVariableDeclaration.name().accept(this);
    if (singleVariableDeclaration.optionalInitialization()
        instanceof
        Make_TM_OptionalExpression) {
      result.append(" ");
      singleVariableDeclaration.optionalInitialization().accept(
          this);
    }
  }

  public void visitMake_TM_OptionalBlock(Make_TM_OptionalBlock
      optionalBlock) {
    optionalBlock.block().accept(this);
  }

  public void visitTM_NoOptionalBlock(TM_NoOptionalBlock
      noOptionalBlock) {
    result.append(";");
  }

  public void visitTM_Block(TM_Block block) {
    result.append("{\n");
    for (TM_Statement statement : block.statementList().getList
        ()) {
      statement.accept(this);
      result.append("\n");
    }
    result.append("}");
  }

  public void visitTM_ExpressionStatement(TM_ExpressionStatement
                                                  expressionStatement) {
    expressionStatement.expression().accept(this);
    result.append(";");
  }

  public void visitTM_IfStatement(TM_IfStatement ifStatement) {
    result.append("if(");
    ifStatement.condition().accept(this);
    result.append(")");
    ifStatement.ifBlock().accept(this);
```

```
    if (ifStatement.elseBlock() instanceof Make_TM_OptionalBlock
        ) {
      result.append("\nelse");
      ifStatement.elseBlock().accept(this);
    }
  }

  public void visitTM_ReturnStatement(TM_ReturnStatement
      returnStatement) {
    result.append("return ");
    returnStatement.expressionReturnStatement().accept(this);
    result.append(";");
  }

  public void visitMake_TM_VariableDeclarationStatement(
      Make_TM_VariableDeclarationStatement
          makeVariableDeclarationStatement) {
    makeVariableDeclarationStatement.
        variableDeclarationStatement().accept(this);
  }

  public void visitTM_VariableDeclarationStatement(
      TM_VariableDeclarationStatement
          variableDeclarationStatement) {
    for (TM_Modifier modifier :
        variableDeclarationStatement.modifierList().getList())
            {
      modifier.accept(this);
      result.append(" ");
    }
    variableDeclarationStatement.getType().accept(this);
    result.append(" ");
    variableDeclarationStatement.fragment().accept(this);
    result.append(";");
  }

  public void visitTM_VariableDeclarationFragment(
      TM_VariableDeclarationFragment variableDeclarationFragment
          ) {
    variableDeclarationFragment.name().accept(this);
    if (variableDeclarationFragment.optionalExpression()
        instanceof
        Make_TM_OptionalExpression) {
      result.append(" = ");
      variableDeclarationFragment.optionalExpression().accept(
          this);
    }
  }

```

```
    /*Expressions*/
    public void visitMake_TM_OptionalExpression(
        Make_TM_OptionalExpression
                                                  makeOptionalExpression
                                                    ) {
      makeOptionalExpression.expression().accept(this);
295 }

    public void visitTM_NoOptionalExpression(
        TM_NoOptionalExpression
                                                noOptionalExpression)
                                                  {}

300 public void visitMake_TM_JavaValueLiteral(
        Make_TM_JavaValueLiteral
                                                 makeValueLiteral) {
      makeValueLiteral.valueLiteral().accept(this);
    }

305 public void visitMake_TM_JavaValueLiteralInteger(
         Make_TM_JavaValueLiteralInteger makeValueLiteralInteger) {
      makeValueLiteralInteger.valueLiteralInteger().accept(this);
    }

310 public void visitTM_JavaValueLiteralInteger(
        TM_JavaValueLiteralInteger
                                                  valueLiteralInteger
                                                    ) {
      result.append(valueLiteralInteger.getText());
    }

315 public void visitMake_TM_JavaValueLiteralString(
         Make_TM_JavaValueLiteralString makeValueLiteralString) {
      makeValueLiteralString.valueLiteralString().accept(this);
    }

320 public void visitTM_JavaValueLiteralString(
        TM_JavaValueLiteralString
                                                valueLiteralString)
                                                  {
      result.append("\"" + valueLiteralString.getText() + "\"");
    }

325 public void visitMake_TM_JavaValueLiteralChar(
        Make_TM_JavaValueLiteralChar
                                                    makeValueLiteralChar
                                                      ) {
      makeValueLiteralChar.valueLiteralChar().accept(this);
    }
```

```
330   public void visitTM_JavaValueLiteralChar(
          TM_JavaValueLiteralChar
                                                valueLiteralChar) {
        result.append("\'" + valueLiteralChar.getText() + "\'");
      }

335   public void visitMake_TM_JavaValueLiteralDouble(
           Make_TM_JavaValueLiteralDouble makeValueLiteralDouble) {
        makeValueLiteralDouble.valueLiteralDouble().accept(this);
      }

340   public void visitTM_JavaValueLiteralDouble(
          TM_JavaValueLiteralDouble
                                                  valueLiteralDouble)
                                                     {
        result.append(valueLiteralDouble.getText());
      }

345   public void visitMake_TM_JavaValueLiteralBool(
          Make_TM_JavaValueLiteralBool
                                                    makeValueLiteralBool
                                                       ) {
        makeValueLiteralBool.valueLiteralBool().accept(this);
      }

350   public void visitTM_JavaValueLiteralBool(
          TM_JavaValueLiteralBool
                                                valueLiteralBool) {
        result.append(valueLiteralBool.getText());
      }

355   public void visitTM_NullLiteral(TM_NullLiteral nullLiteral) {
        result.append("null");
      }

      public void visitTM_ArrayCreation(TM_ArrayCreation
          arrayCreation) {
360     result.append("new ");
        arrayCreation.getArrayType().accept(this);
        result.append("[");
        if (arrayCreation.getCount() instanceof
            Make_TM_OptionalExpression) {
          arrayCreation.getCount().accept(this);
365     }
        result.append("]");
        if (arrayCreation.elementList().len() > 0) {
          result.append("{");
          for (TM_Expression expression : arrayCreation.elementList
```

```
            ().getList()) {
370           expression.accept(this);
              result.append(", ");
            }
            if (arrayCreation.elementList().len() > 0) {
              result.delete(result.length() - 2, result.length());
375         }
            result.append("}");
          }
        }

380     public void visitTM_InfixExpression(TM_InfixExpression
            infixExpression) {
          infixExpression.left().accept(this);
          result.append(" ");
          infixExpression.op().accept(this);
          result.append(" ");
385       infixExpression.right().accept(this);
        }

        public void visitTM_PrefixExpression(TM_PrefixExpression
            prefixExpression) {
          prefixExpression.prefixOperator().accept(this);
390       prefixExpression.prefixExpression().accept(this);
        }

        public void visitTM_MethodInvocation(TM_MethodInvocation
            methodInvocation) {
          methodInvocation.optionalExpression().accept(this);
395       if (methodInvocation.optionalExpression() instanceof
                Make_TM_OptionalExpression) {
            result.append(".");
          }
          methodInvocation.name().accept(this);
400       result.append("(");
          for (TM_Expression expression : methodInvocation.
              argumentList().getList()) {
            expression.accept(this);
            result.append(", ");
          }
405       if (methodInvocation.argumentList().len() > 0) {
            result.delete(result.length() - 2, result.length());
          }
          result.append(")");
        }
410

        public void visitTM_ClassInstanceCreation(
            TM_ClassInstanceCreation
                                                  classInstanceCreation
```

```
                                                    ) {
    if (classInstanceCreation.
        optionalExpressionClassInstanceCreation() instanceof
         Make_TM_OptionalExpression) {
      classInstanceCreation.
          optionalExpressionClassInstanceCreation().accept(this);
      result.append(".");
    }
    result.append("new ");
    if (classInstanceCreation.getType() != null) {
      classInstanceCreation.getType().accept(this);
    }
    else {
      result.append("
          UNKNOWN_TYPE_INSERTED_BY_CLASSINSTANCECREATION");
    }
    result.append("(");
    for (TM_Expression expression :
         classInstanceCreation.argumentListClassInstanceCreation
             ().getList()) {
      expression.accept(this);
      result.append(", ");
    }
    if (classInstanceCreation.argumentListClassInstanceCreation
        ().len() > 0) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")");
  }

  public void visitTM_AssignmentExpression(
      TM_AssignmentExpression
                                           assignmentExpression)
                                              {
    assignmentExpression.lhs().accept(this);
    result.append(" ");
    assignmentExpression.assignmentOp().accept(this);
    result.append(" ");
    assignmentExpression.rhs().accept(this);
  }

  public void visitTM_ParenthesizedExpression(
      TM_ParenthesizedExpression
                                                 parenthesizedExpression
                                                    ) {
    result.append("(");
    parenthesizedExpression.expression().accept(this);
    result.append(")");
  }
```

```
  public void visitTM_InstanceOfExpression(
      TM_InstanceOfExpression
                                              instanceOfExpression)
                                                  {
455   instanceOfExpression.instanceOfExpression().accept(this);
    result.append(" instanceof ");
    instanceOfExpression.instanceOfType().accept(this);
  }

460 public void visitTM_ThisExpression(TM_ThisExpression
      thisExpression) {
    if (thisExpression.thisName() instanceof
        Make_TM_OptionalSimpleName) {
      thisExpression.thisName().accept(this);
      result.append(".");
    }
465 result.append("this");
  }

  public void visitTM_CastExpression(TM_CastExpression
      castExpression) {
    result.append("(");
470 castExpression.castType().accept(this);
    result.append(")");
    castExpression.castExpression().accept(this);
  }

475 public void visitTM_FieldAccessExpression(
      TM_FieldAccessExpression
                                            fieldAccessExpression
                                                ) {
    if (fieldAccessExpression.fieldExpression() instanceof
        Make_TM_OptionalExpression) {
      fieldAccessExpression.fieldExpression().accept(this);
480   result.append(".");
    }
    fieldAccessExpression.fieldName().accept(this);
  }

485 public void visitTM_JAVA_NOT(TM_JAVA_NOT not) {
    result.append("!");
  }

  public void visitTM_JAVA_EQUALS(TM_JAVA_EQUALS equals) {
490 result.append("==");
  }

  public void visitTM_JAVA_PLUS(TM_JAVA_PLUS plus) {
```

```
    result.append("+");
  }

  public void visitTM_JAVA_STAR(TM_JAVA_STAR star) {
    result.append("*");
  }

  public void visitTM_JAVA_DIV(TM_JAVA_DIV div) {
    result.append("/");
  }

  public void visitTM_JAVA_ASSIGNMENT_OP_EQUAL(
      TM_JAVA_ASSIGNMENT_OP_EQUAL
                                                equal) {
    result.append("=");
  }

  /*NAMES*/
  public void visitMake_TM_JavaName(Make_TM_JavaName
      makeJavaName) {
    makeJavaName.name().accept(this);
  }

  public void visitMake_TM_SimpleName(Make_TM_SimpleName
      makeSimpleName) {
    makeSimpleName.name().accept(this);
  }

  public void visitMake_TM_QualifiedName(Make_TM_QualifiedName
                                           makeQualifiedName) {
    makeQualifiedName.name().accept(this);
  }

  public void visitTM_QualifiedName(TM_QualifiedName
      qualifiedName) {
    qualifiedName.left().accept(this);
    result.append(".");
    qualifiedName.right().accept(this);
  }

  public void visitTM_SimpleName(TM_SimpleName simpleName) {
    result.append(simpleName.text());
  }

  public void visitTM_NoOptionalSimpleName(
      TM_NoOptionalSimpleName
                                             noOptionalSimpleName)
                                               {}
```

```
    public void visitMake_TM_OptionalSimpleName(
        Make_TM_OptionalSimpleName
                                                optionalSimpleName
                                                    ) {
      optionalSimpleName.name().accept(this);
    }

    /*Types*/
    public void visitMake_TM_ReferenceType(Make_TM_ReferenceType
                                                makeReferenceType) {
      makeReferenceType.referenceType().accept(this);
    }

    public void visitMake_TM_ArrayType(Make_TM_ArrayType
        makeArrayType) {
      makeArrayType.arrayType().accept(this);
    }

    public void visitMake_TM_PrimitiveType(Make_TM_PrimitiveType
                                                makePrimitiveType) {
      makePrimitiveType.primitiveType().accept(this);
    }

    public void visitTM_ReferenceType(TM_ReferenceType
        referenceType) {
      referenceType.name().accept(this);
      if (referenceType.optionalTypeArgument() instanceof
          Make_TM_OptionalReferenceType) {
        result.append("<");
        referenceType.optionalTypeArgument().accept(this);
        result.append(">");
      }
    }

    public void visitTM_ArrayType(TM_ArrayType arrayType) {
      arrayType.getType().accept(this);
      result.append("[]");
    }

    public void visitMake_TM_OptionalReferenceType(
        Make_TM_OptionalReferenceType
                                                optionalReferenceType
                                                    ) {
      optionalReferenceType.referenceType().accept(this);
    }

    public void visitTM_NoOptionalReferenceType(
        TM_NoOptionalReferenceType
                                                noOptionalReferenceType
```

```
                                                    ) {}

  public void visitTM_JAVA_INT(TM_JAVA_INT javaInt) {
    result.append("int");
  }

  public void visitTM_JAVA_VOID(TM_JAVA_VOID javaVoid) {
    result.append("void");
  }

  public void visitTM_JAVA_DOUBLE(TM_JAVA_DOUBLE java_double) {
    result.append("double");
  }

  public void visitTM_JAVA_BOOL(TM_JAVA_BOOL java_bool) {
    result.append("boolean");
  }

  public void visitTM_JAVA_CHAR(TM_JAVA_CHAR java_char) {
    result.append("char");
  }

  /*Other*/
  public void visitTM_JAVA_PUBLIC(TM_JAVA_PUBLIC java_public) {
    result.append("public");
  }

  public void visitTM_JAVA_STATIC(TM_JAVA_STATIC java_static) {
    result.append("static");
  }

  public void visitTM_JAVA_ABSTRACT(TM_JAVA_ABSTRACT
      java_abstract) {
    result.append("abstract");
  }

  public void visitTM_JAVA_PRIVATE(TM_JAVA_PRIVATE java_private)
      {
    result.append("private");
  }
}
```

F.2.6 TM_StringRSLAstVisitor.java

```
package translator.lib;

import translator.rslast2.*;

public class TM_StringRSLAstVisitor
```

```
    extends TM_RSLAstVisitor {
  private StringBuffer result;

  public TM_StringRSLAstVisitor() {
    this.result = new StringBuffer();
  }

  public String getResult() {
    return result.toString();
  }

  public void visitTM_RSLAst(TM_RSLAst rslast) {
    rslast.libmodule().accept(this);
  }

  public void visitTM_LibModule(TM_LibModule module) {
    for (TM_Id id : module.context_list().getList()) {
      id.accept(this);
      result.append(", ");
    }
    if (module.context_list().len() > 0) {
      result.delete(result.length() - 2, result.length());
    }
    result.append("\n");
    module.schemedef().accept(this);
  }

  public void visitTM_SchemeDef(TM_SchemeDef scheme) {
    result.append("scheme ");
    scheme.id().accept(this);
    result.append(" = ");
    scheme.class_expr().accept(this);
  }

  public void visitTM_ExtendingClassExpr(TM_ExtendingClassExpr
                                         extendingClassExpr) {
    result.append(" extend ");
    extendingClassExpr.base_class().accept(this);
    result.append(" with ");
    extendingClassExpr.extension_class().accept(this);
  }

  public void visitTM_SchemeInstantiation(TM_SchemeInstantiation
                                          schemeInstantiation) {
    schemeInstantiation.id().accept(this);
  }

  public void visitTM_BasicClassExpr(TM_BasicClassExpr
      basicClassExpr) {
```

```
    result.append("class\n");
    for (TM_Decl decl : basicClassExpr.declaration_list().
        getList()) {
      decl.accept(this);
    }
    result.append("end");
  }

  /*Type Declarations */
  public void visitTM_TypeDecl(TM_TypeDecl typeDecl) {
    result.append("type\n");
    for (TM_TypeDef typeDef : typeDecl.type_def_list().getList()
        ) {
      typeDef.accept(this);
      result.append(",\n");
    }
    if (typeDecl.type_def_list().len() > 0) {
      result.delete(result.length() - 2, result.length());
    }
    result.append("\n");
  }

  public void visitTM_SortDef(TM_SortDef sortDef) {
    sortDef.sd_id().accept(this);
  }

  public void visitTM_ShortRecordDef(TM_ShortRecordDef
      shortRecordDef) {
    shortRecordDef.srd_id().accept(this);
    result.append(" :: ");

    for (TM_ComponentKind componentKind :
        shortRecordDef.component_kind_string().getList()) {
      componentKind.accept(this);
      result.append(" ");
    }
  }

  public void visitTM_VariantDef(TM_VariantDef variantDef) {
    variantDef.id().accept(this);
    result.append(" == ");
    for (TM_Variant variant : variantDef.variant_list().getList
        ()) {
      variant.accept(this);
      result.append(" | ");
    }
    if (variantDef.variant_list().len() > 0) {
      result.delete(result.length() - 3, result.length());
    }
```

```
    }

    public void visitTM_RecordVariant(TM_RecordVariant
        recordVariant) {
      recordVariant.record_constructor().accept(this);
      result.append("(");
      for (TM_ComponentKind componentKind :
          recordVariant.component_kind_list().getList()) {
        componentKind.accept(this);
        result.append(", ");
      }
      if (recordVariant.component_kind_list().len() > 0) {
        result.delete(result.length() - 2, result.length());
      }
      result.append(")");
    }

    public void visitTM_ComponentKind(TM_ComponentKind
        componentKind) {
      if (componentKind.optional_destructor() instanceof
          TM_Destructor) {
        componentKind.optional_destructor().accept(this);
        result.append(" : ");
      }
      componentKind.type_expr().accept(this);
      if (componentKind.optional_reconstructor() instanceof
          TM_Reconstructor) {
        result.append(" <-> ");
        componentKind.optional_reconstructor().accept(this);
      }
    }

    public void visitMake_TM_Constructor(Make_TM_Constructor
        constructor) {
      constructor.constructor().accept(this);
    }

    public void visitTM_Constructor(TM_Constructor constructor) {
      constructor.id().accept(this);
    }

    public void visitTM_Destructor(TM_Destructor destructor) {
      destructor.id().accept(this);
    }

    public void visitTM_NoDestructor(TM_NoDestructor noDestructor)
        {}

    public void visitTM_Reconstructor(TM_Reconstructor
```

```
      reconstructor) {
    reconstructor.id().accept(this);
  }

  public void visitTM_NoReconstructor(TM_NoReconstructor
      noReconstructor) {}

  /*Value Declarations*/
  public void visitTM_ValueDecl(TM_ValueDecl valueDecl) {
    result.append("value\n");
    for (TM_ValueDef valueDef : valueDecl.value_def_list().
        getList()) {
      valueDef.accept(this);
      result.append(",\n");
    }
    if (valueDecl.value_def_list().len() > 0) {
      result.delete(result.length() - 2, result.length());
    }
    result.append("\n");
  }

  public void visitTM_ExplicitFunctionDef(TM_ExplicitFunctionDef
                                          explicitFunctionDef) {
    explicitFunctionDef.single_typing().accept(this);
    result.append("\n");
    explicitFunctionDef.formal_function_application().accept(
        this);
    result.append(" is ");
    explicitFunctionDef.value_expr().accept(this);
  }

  public void visitTM_SingleTyping(TM_SingleTyping singleTyping)
      {
    singleTyping.binding().accept(this);
    result.append(" : ");
    singleTyping.type_expr().accept(this);
  }

  public void visitTM_IdApplication(TM_IdApplication
      idApplication) {
    idApplication.id().accept(this);
    result.append("(");
    for (TM_FormalFunctionParameter parameter :
        idApplication.formal_function_parameter_list().getList
            ()) {
      parameter.accept(this);
      result.append(", ");
    }
    if (idApplication.formal_function_parameter_list().len()
```

```
        > 0) {
      result.delete(result.length() - 2, result.length());
    }
    result.append(")");
  }

  public void visitTM_FormalFunctionParameter(
      TM_FormalFunctionParameter
                                                formalFunctionParameter
                                                    ) {
    for (TM_Binding binding : formalFunctionParameter.
        binding_list().getList()) {
      binding.accept(this);
      result.append(", ");
    }
    if (formalFunctionParameter.binding_list().len() > 0) {
      result.delete(result.length() - 2, result.length());
    }
  }

  /*Test Declarations*/
  public void visitTM_TestDecl(TM_TestDecl testDecl) {
    for (TM_TestDef testDef : testDecl.test_def_list().getList()
        ) {
      testDef.accept(this);
      result.append(",\n");
    }
    if (testDecl.test_def_list().len() > 0) {
      result.delete(result.length() - 2, result.length());
    }
    result.append("\n");
  }

  public void visitTM_TestCase(TM_TestCase testCase) {
    result.append("[");
    testCase.id().accept(this);
    result.append("] ");
    testCase.value_expr().accept(this);
  }

  /*Type Expression*/
  public void visitTM_TypeExprList(TM_TypeExprList typeExprList)
      {
    typeExprList.type_expr_list().accept(this);
  }

  public void visitTM_FiniteListTypeExpr(TM_FiniteListTypeExpr
                                           finiteListTypeExpr) {
    finiteListTypeExpr.type_expr().accept(this);
```

```
    result.append("-list");
  }

  public void visitTM_FunctionTypeExpr(TM_FunctionTypeExpr
      functionTypeExpr) {
    functionTypeExpr.type_expr_argument().accept(this);
    result.append(" ");
    functionTypeExpr.function_arrow().accept(this);
    result.append(" ");
    functionTypeExpr.type_expr_result().accept(this);
  }

  public void visitTM_TOTAL_FUNCTION_ARROW(
      TM_TOTAL_FUNCTION_ARROW
                                          totalFunctionArrow) {
    result.append("->");
  }

  public void visitTM_TypeExprProduct(TM_TypeExprProduct
      typeExprProduct) {
    for (TM_TypeExpr te : typeExprProduct.component_list().
        getList()) {
      te.accept(this);
      result.append(" >< ");
    }
    if (typeExprProduct.component_list().len() > 0) {
      result.delete(result.length() - 4, result.length());
    }
  }

  public void visitTM_TypeName(TM_TypeName typeName) {
    typeName.id().accept(this);
  }

  public void visitTM_TypeLiteral(TM_TypeLiteral typeLiteral) {
    typeLiteral.type_literal().accept(this);
  }

  public void visitTM_RSL_UNIT(TM_RSL_UNIT rsl_unit) {
    result.append("Unit");
  }

  public void visitTM_RSL_INT(TM_RSL_INT tm_rsl_int) {
    result.append("Int");
  }

  public void visitTM_RSL_NAT(TM_RSL_NAT rsl_nat) {
    result.append("Nat");
  }
```

```
    public void visitTM_RSL_REAL(TM_RSL_REAL rsl_real) {
      result.append("Real");
    }

    public void visitTM_RSL_BOOL(TM_RSL_BOOL rsl_bool) {
      result.append("Bool");
    }

    public void visitTM_RSL_CHAR(TM_RSL_CHAR rsl_char) {
      result.append("Char");
    }

    public void visitTM_RSL_TEXT(TM_RSL_TEXT rsl_text) {
      result.append("Text");
    }

    /*Value Expression*/
    public void visitMake_TM_ListExpr(Make_TM_ListExpr
        makeListExpr) {
      makeListExpr.list_expr().accept(this);
    }

    public void visitTM_EnumeratedListExpr(TM_EnumeratedListExpr
                                           enumeratedListExpr) {
      result.append("<.");
      for (TM_ValueExpr ve : enumeratedListExpr.value_expr_list().
          getList()) {
        ve.accept(this);
        result.append(", ");
      }
      if (enumeratedListExpr.value_expr_list().len() > 0) {
        result.delete(result.length() - 2, result.length());

      }
      result.append(".>");
    }

    public void visitTM_ApplicationExpr(TM_ApplicationExpr
        applicationExpr) {
      applicationExpr.value_expr().accept(this);
      result.append("(");
      for (TM_ValueExpr ve : applicationExpr.value_expr_list().
          getList()) {
        ve.accept(this);
        result.append(", ");
      }
      if (applicationExpr.value_expr_list().len() > 0) {
        result.delete(result.length() - 2, result.length());
```

```
    }
    result.append(")");
  }

  public void visitMake_TM_IfExpr(Make_TM_IfExpr makeIfExpr) {
    makeIfExpr.if_expr().accept(this);
  }

  public void visitTM_IfExpr(TM_IfExpr ifExpr) {
    result.append("if ");
    ifExpr.condition().accept(this);
    result.append(" then ");
    ifExpr.if_case().accept(this);
    for (TM_Elsif elsif : ifExpr.elsif_list().getList()) {
      elsif.accept(this);
    }
    result.append(" else ");
    ifExpr.else_case().accept(this);
    result.append(" end");
  }

  public void visitTM_Elsif(TM_Elsif elsif) {
    result.append("elsif ");
    elsif.condition().accept(this);
    result.append(" then ");
    elsif.elsif_case().accept(this);
  }

  public void visitTM_ValueInfixExpr(TM_ValueInfixExpr
      valueInfixExpr) {
    valueInfixExpr.left().accept(this);
    result.append(" ");
    valueInfixExpr.op().accept(this);
    result.append(" ");
    valueInfixExpr.right().accept(this);
  }

  public void visitTM_CaseExpr(TM_CaseExpr caseExpr) {
    result.append("case ");
    caseExpr.condition().accept(this);
    result.append(" of\n");
    for (TM_CaseBranch cb : caseExpr.case_branch_list().getList
        ()) {
      cb.accept(this);
      result.append(",\n");
    }
    if (caseExpr.case_branch_list().len() > 0) {
      result.delete(result.length() - 2, result.length());
    }
```

```
    result.append("\n");
    result.append("end");
  }

  public void visitTM_CaseBranch(TM_CaseBranch caseBranch) {
    caseBranch.pattern().accept(this);
    result.append(" -> ");
    caseBranch.value_expr().accept(this);
  }

  public void visitTM_ValuePrefixExpr(TM_ValuePrefixExpr
      valuePrefixExpr) {
    valuePrefixExpr.op().accept(this);
    result.append(" ");
    valuePrefixExpr.operand().accept(this);
  }

  public void visitTM_ParenthesizedExpr(TM_ParenthesizedExpr
      parenthesizedExpr) {
    result.append("(");
    parenthesizedExpr.parenthesized_expr().accept(this);
    result.append(")");
  }

  public void visitTM_RSL_EQUAL(TM_RSL_EQUAL rsl_equal) {
    result.append("=");
  }

  public void visitTM_RSL_PLUS(TM_RSL_PLUS rsl_plus) {
    result.append("+");
  }

  public void visitTM_RSL_STAR(TM_RSL_STAR rsl_star) {
    result.append("*");
  }

  public void visitTM_RSL_SLASH(TM_RSL_SLASH rsl_slash) {
    result.append("/");
  }

  public void visitTM_RSL_HAT(TM_RSL_HAT rsl_hat) {
    result.append("^");
  }

  public void visitTM_RSL_HD(TM_RSL_HD rsl_hd) {
    result.append("hd");
  }

  public void visitTM_RSL_TL(TM_RSL_TL rsl_tl) {
```

```
    result.append("tl");
  }

  public void visitTM_ValueOrVariableName(TM_ValueOrVariableName
                                          valueOrVariableName) {
    valueOrVariableName.id().accept(this);
  }

  public void visitTM_ValueLiteralInteger(TM_ValueLiteralInteger
                                          valueLiteralInteger) {
    result.append(valueLiteralInteger.getTextInteger());
  }

  public void visitTM_ValueLiteralReal(TM_ValueLiteralReal
      valueLiteralReal) {
    result.append(valueLiteralReal.getTextReal());
  }

  public void visitTM_ValueLiteralBool(TM_ValueLiteralBool
      valueLiteralBool) {
    result.append(valueLiteralBool.getTextBool());
  }

  public void visitTM_ValueLiteralChar(TM_ValueLiteralChar
      valueLiteralChar) {
    result.append("'");
    result.append(valueLiteralChar.getTextChar());
    result.append("'");
  }

  public void visitTM_ValueLiteralText(TM_ValueLiteralText
      valueLiteralText) {
    result.append("\"");
    result.append(valueLiteralText.getTextText());
    result.append("\"");
  }

  /*Common*/
  public void visitTM_Binding(TM_Binding binding) {
    binding.id().accept(this);
  }

  public void visitTM_Id(TM_Id id) {
    result.append(id.getText());
  }
}
```

F.2.7 TM_TypeDecorateRSLAstVisitor.java

```
package translator.lib;

import translator.*;
import translator.rslast2.*;
import translator.rsllib.*;

import java.util.*;

public class TM_TypeDecorateRSLAstVisitor extends
    TM_RSLAstVisitor {
    /*Map of function names to a map of identifiers to types of
        identifiers*/
    private HashMap<String, HashMap<String, TM_TypeEvaluator>>
        map;
    /*Map of types of identifiers of function currently being
        evaluated*/
    private HashMap<String, TM_TypeEvaluator> currentMap;
    /*List of types of components of record constructor
        currently evaluated*/
    private ArrayList<TM_TypeEvaluator> currentRecordType;
    /*List of id of record constructors currently evaluated*/
    private ArrayList<String> currentRecordId;
    /*Map from record constructor names to a list of types of
        the components*/
    private HashMap<String, ArrayList<TM_TypeEvaluator>>
        recordMapType;
    /*Map from record constructor names to a list of names of
        the destructors*/
    private HashMap<String, ArrayList<String>> recordMapId;
    /*Map from function names to a list of the parameters
        overlaps with map*/
    private HashMap<String, ArrayList<TM_TypeEvaluator>>
        functionMapType;
    /*Map from function names to result of a function*/
    private HashMap<String, TM_TypeEvaluator> functionResultType
        ;
    /*Map from alternatives of a variant definition to the
        varian definition name*/
    private HashMap<String, String> parentMap;
    /*List of method which must be translated as dynamic method
        invocations*/
    private ArrayList<String> ooExternalMethodList;
    /*Variable determining if debug information should be
        written or not*/
    private boolean debug;

    public TM_TypeDecorateRSLAstVisitor() {
        this.map = new HashMap<String, HashMap<String,
            TM_TypeEvaluator>>();
```

```
35          this.recordMapType = new HashMap<String, ArrayList<
                TM_TypeEvaluator>>();
            this.recordMapId = new HashMap<String, ArrayList<String
                >>();
            this.functionMapType = new HashMap<String, ArrayList<
                TM_TypeEvaluator>>();
            this.functionResultType = new HashMap<String,
                TM_TypeEvaluator>();
            this.parentMap = new HashMap<String, String>();
40          this.ooExternalMethodList = new ArrayList<String>();
            this.debug = false;
        }

        public TM_TypeDecorateRSLAstVisitor(HashMap<String, HashMap<
            String, TM_TypeEvaluator>> map,
45                                          HashMap<String,
                                                ArrayList<
                                                TM_TypeEvaluator>>
                                                recordMapType,
                                            HashMap<String,
                                                ArrayList<String>>
                                                recordMapId,
                                            HashMap<String,
                                                ArrayList<
                                                TM_TypeEvaluator>>
                                                functionMapType,
                                            HashMap<String,
                                                TM_TypeEvaluator>
                                                functionResultType,
                                            HashMap<String, String>
                                                parentMap,
50                                          ArrayList<String>
                                                ooExternalMethodList,
                                            boolean debug
                                            ) {
            this.map = map;
            this.recordMapType = recordMapType;
55          this.recordMapId = recordMapId;
            this.functionMapType = functionMapType;
            this.functionResultType = functionResultType;
            this.parentMap = parentMap;
            this.ooExternalMethodList = ooExternalMethodList;
60          this.debug = debug;
        }

        public void visitTM_RSLAst(TM_RSLAst rslast) {
            rslast.libmodule().accept(this);
65      }
```

```
    public void visitTM_LibModule(TM_LibModule module) {
        module.schemedef().accept(this);
    }

    public void visitTM_SchemeDef(TM_SchemeDef scheme) {
        scheme.id().accept(this);
        scheme.class_expr().accept(this);
    }

    public void visitTM_BasicClassExpr(TM_BasicClassExpr
        basicClassExpr) {
        for(TM_Decl decl : basicClassExpr.declaration_list().
            getList()) {
            decl.accept(this);
        }
    }

    /*Type Declarations */
    public void visitTM_TypeDecl(TM_TypeDecl typeDecl) {
        for(TM_TypeDef typeDef : typeDecl.type_def_list().
            getList()) {
            typeDef.accept(this);
        }
    }


    public void visitTM_SortDef(TM_SortDef sortDef) {
        sortDef.sd_id().accept(this);
    }


    public void visitTM_VariantDef(TM_VariantDef variantDef) {
        variantDef.id().accept(this);
        for(TM_Variant variant : variantDef.variant_list().
            getList()) {
            variant.accept(this);
        }
    }

    public void visitTM_ShortRecordDef(TM_ShortRecordDef
        shortRecordDef) {
        this.currentMap = new HashMap<String, TM_TypeEvaluator
            >();
        this.currentRecordType = new ArrayList<TM_TypeEvaluator
            >();
        this.currentRecordId = new ArrayList<String>();

        shortRecordDef.srd_id().accept(this);
        for(TM_ComponentKind componentKind : shortRecordDef.
```

```
            component_kind_string().getList()) {
             componentKind.accept(this);
        }

        this.currentMap.remove(shortRecordDef.srd_id().getText()
            );
        this.currentMap.put(shortRecordDef.srd_id().getText(),
            new TM_TypeEvaluator_TM_TypeExpr(new TM_TypeName(
            shortRecordDef.srd_id())));

        this.parentMap.put(shortRecordDef.srd_id().getText(),
            shortRecordDef.srd_id().getText());
        this.recordMapType.put(shortRecordDef.srd_id().getText()
            , this.currentRecordType);
        this.recordMapId.put(shortRecordDef.srd_id().getText(),
            this.currentRecordId);
        this.map.put(shortRecordDef.srd_id().getText(), this.
            currentMap);
    }

    public void visitTM_Constructor(TM_Constructor constructor)
        {
        if(!(constructor.getParent() instanceof TM_RecordVariant
            ))
            this.currentMap = new HashMap<String,
                TM_TypeEvaluator>();

        constructor.id().accept(this);
        this.currentMap.remove(constructor.id().getText());
        //this.currentMap.put(constructor.getId(), constructor);
        this.currentMap.put(constructor.id().getText(), new
            TM_TypeEvaluator_TM_Constructor(constructor));

        if(constructor.getParent() instanceof TM_VariantDef)
            this.parentMap.put(constructor.id().getText(), ((
                TM_VariantDef) constructor.getParent()).id().
                getText());

        if(!(constructor.getParent() instanceof TM_RecordVariant
            ))
            this.map.put(constructor.id().getText(), this.
                currentMap);
    }

    public void visitTM_Destructor(TM_Destructor destructor) {
        destructor.id().accept(this);
    }

    public void visitTM_RecordVariant(TM_RecordVariant
```

```
        recordVariant) {
        this.currentMap = new HashMap<String, TM_TypeEvaluator
            >();
        this.currentRecordType = new ArrayList<TM_TypeEvaluator
            >();
        this.currentRecordId = new ArrayList<String>();

        recordVariant.record_constructor().accept(this);
        for(TM_ComponentKind componentKind : recordVariant.
            component_kind_list().getList()) {
            componentKind.accept(this);
        }

        //System.out.println("Parent: " + recordVariant.
            getParent());

        this.parentMap.put(recordVariant.record_constructor().id
            ().getText(), ((TM_VariantDef) recordVariant.
            getParent()).id().getText());
        this.recordMapType.put(recordVariant.record_constructor
            ().id().getText(), this.currentRecordType);
        this.recordMapId.put(recordVariant.record_constructor().
            id().getText(), this.currentRecordId);
        this.map.put(recordVariant.record_constructor().id().
            getText(), this.currentMap);
    }

    public void visitTM_ComponentKind(TM_ComponentKind
        componentKind) {
        componentKind.type_expr().accept(this);
        componentKind.optional_destructor().accept(this);
        if(componentKind.optional_destructor() instanceof
            TM_Destructor) {
            this.currentMap.remove(((TM_Destructor)componentKind
                .optional_destructor()).id().getText());
            this.currentMap.put(((TM_Destructor)componentKind.
                optional_destructor()).id().getText(), new
                TM_TypeEvaluator_TM_Destructor(componentKind.
                optional_destructor()));
            this.currentRecordType.add(new
                TM_TypeEvaluator_TM_TypeExpr(componentKind.
                type_expr()));
            this.currentRecordId.add(((TM_Destructor)
                componentKind.optional_destructor()).id().getText
                ());
            //Added
            this.map.put(((TM_Destructor)componentKind.
                optional_destructor()).id().getText(), this.
                currentMap);
```

```
        }
170         componentKind.optional_reconstructor().accept(this);
    }

    /*Value Declarations*/
    public void visitTM_ValueDecl(TM_ValueDecl valueDecl) {
175         for(TM_ValueDef valueDef : valueDecl.value_def_list().
                getList()) {
             valueDef.accept(this);
        }
    }

180     public void visitTM_ExplicitFunctionDef(
            TM_ExplicitFunctionDef explicitFunctionDef) {
         this.currentMap = new HashMap<String, TM_TypeEvaluator
             >();
         this.currentRecordType = new ArrayList<TM_TypeEvaluator
             >();


185          explicitFunctionDef.single_typing().accept(this);
         TM_Id functionId = explicitFunctionDef.single_typing().
             binding().id();
         explicitFunctionDef.formal_function_application().accept
             (this);



190
         RSLList<TM_TypeEvaluator> parameterList = null;
         if(explicitFunctionDef.single_typing().type_expr()
             instanceof TM_FunctionTypeExpr) {
          TM_TypeExpr parameters = ((TM_FunctionTypeExpr)
              explicitFunctionDef.single_typing().type_expr()).
              type_expr_argument();
          parameterList = new RSLListDefault<TM_TypeEvaluator
              >();
195           if(parameters instanceof TM_TypeExprProduct) {
              TM_TypeExprProduct pte = (TM_TypeExprProduct)
                  parameters;
              for(TM_TypeExpr te : pte.component_list().
                  getList()) {
               parameterList.getList().add(new
                   TM_TypeEvaluator_TM_TypeExpr(te));
               //System.out.println("Adding a product part
                   : " + te);
200                this.currentRecordType.add(new
                   TM_TypeEvaluator_TM_TypeExpr(te));
              }
          }
```

```
            else if(parameters instanceof TM_TypeExprList) {
                parameterList.getList().add(new
                    TM_TypeEvaluator_TM_TypeExpr(parameters));
205             this.currentRecordType.add(new
                    TM_TypeEvaluator_TM_TypeExpr(parameters));
                //System.out.println("Adding: " + parameters);
            }
            else if(parameters instanceof TM_TypeLiteral) {
                parameterList.getList().add(new
                    TM_TypeEvaluator_TM_TypeExpr(parameters));
210             this.currentRecordType.add(new
                    TM_TypeEvaluator_TM_TypeExpr(parameters));
                //System.out.println("Adding " + parameters);
            }
            else if(parameters instanceof TM_TypeName) {
                parameterList.getList().add(new
                    TM_TypeEvaluator_TM_TypeExpr(parameters));
215             this.currentRecordType.add(new
                    TM_TypeEvaluator_TM_TypeExpr(parameters));
            }
        }
        if(explicitFunctionDef.formal_function_application()
            instanceof TM_IdApplication) {
            TM_IdApplication ia = (TM_IdApplication)
                explicitFunctionDef.formal_function_application()
                ;
220         for(TM_FormalFunctionParameter ffp : ia.
                formal_function_parameter_list().getList()) {
                for(TM_Binding b : ffp.binding_list().getList())
                    {
                    currentMap.put((b.id().getText()),
                        parameterList.hd());
                    parameterList = parameterList.tl();
                }
225         }
        }
        this.map.put(functionId.getText(), currentMap);
        this.functionMapType.put(functionId.getText(), this.
            currentRecordType);


230
        explicitFunctionDef.setTypeEvaluatorForTypeEvaluation(
            new TM_TypeEvaluator_TM_TypeExpr(((
            TM_FunctionTypeExpr) explicitFunctionDef.
            single_typing().type_expr()).type_expr_result()));
        explicitFunctionDef.value_expr().accept(this);
    }

235 public void visitTM_SingleTyping(TM_SingleTyping
```

```
        singleTyping) {
         singleTyping.binding().accept(this);
         singleTyping.type_expr().accept(this);

         this.functionResultType.put(singleTyping.binding().id().
             getText(), new TM_TypeEvaluator_TM_TypeExpr(((
             TM_FunctionTypeExpr) singleTyping.type_expr()).
             type_expr_result()));
240  }

     public void visitTM_IdApplication(TM_IdApplication
         idApplication) {
         idApplication.id().accept(this);
         for(TM_FormalFunctionParameter parameter : idApplication
             .formal_function_parameter_list().getList()) {
245          parameter.accept(this);
         }
     }

     public void visitTM_FormalFunctionParameter(
         TM_FormalFunctionParameter formalFunctionParameter) {
250      for(TM_Binding binding : formalFunctionParameter.
             binding_list().getList()) {
             binding.accept(this);
         }
     }

255  /*Type Expression*/
     public void visitTM_TypeExprList(TM_TypeExprList
         typeExprList) {
         typeExprList.type_expr_list().accept(this);
     }

260  public void visitTM_FiniteListTypeExpr(TM_FiniteListTypeExpr
          finiteListTypeExpr) {
         finiteListTypeExpr.type_expr().accept(this);
     }

     public void visitTM_FunctionTypeExpr(TM_FunctionTypeExpr
         functionTypeExpr) {
265      functionTypeExpr.type_expr_argument().accept(this);
         functionTypeExpr.function_arrow().accept(this);
         functionTypeExpr.type_expr_result().accept(this);
     }

270  public void visitTM_TOTAL_FUNCTION_ARROW(
         TM_TOTAL_FUNCTION_ARROW totalFunctionArrow) {}

     public void visitTM_TypeLiteral(TM_TypeLiteral typeLiteral)
```

```
        {
        typeLiteral.type_literal().accept(this);
    }

    public void visitTM_TypeName(TM_TypeName typeName) {
        //typeName.id().accept(this);
    }

    public void visitTM_RSL_UNIT(TM_RSL_UNIT rsl_unit) {}

    public void visitTM_RSL_INT(TM_RSL_INT rsl_int) {}

    public void visitTM_RSL_NAT(TM_RSL_NAT rsl_nat) {}

    public void visitTM_RSL_REAL(TM_RSL_REAL rsl_real) {}

    public void visitTM_RSL_BOOL(TM_RSL_BOOL rsl_bool) {}

    public void visitTM_RSL_CHAR(TM_RSL_CHAR rsl_char) {}

    public void visitTM_RSL_TEXT(TM_RSL_TEXT rsl_text) {}

    /*Value Expression*/
    public void visitTM_ApplicationExpr(TM_ApplicationExpr
        applicationExpr) {
        applicationExpr.value_expr().accept(this);
        for(TM_ValueExpr ve : applicationExpr.value_expr_list().
            getList())
            ve.accept(this);

        if(applicationExpr.value_expr() instanceof
            Make_TM_ValueOrVariableName) {
            TM_ValueOrVariableName vovn = ((
                Make_TM_ValueOrVariableName) applicationExpr.
                value_expr()).value_or_variable_name();
            TM_Id id = vovn.id();

            //In case of a short record definition make function
                we need to lookup the short record definition
            if(map.get(id.getText()) == null) {
                if(vovn.id().getText().startsWith("mk_")) {
                    if(map.get(vovn.id().getText().substring(3))
                        != null) {
                        String name = vovn.id().getText().
                            substring(3);
                        //System.out.println("Must be a
                            shortRecordDefinition: " + name + "
                            to: " + (map.get(name)).get(name));
                        applicationExpr.
```

```
                        setTypeEvaluatorForTypeEvaluation((
                        map.get(name)).get(name));
                    applicationExpr.value_expr().
                        setTypeEvaluatorForTypeEvaluation((
                        map.get(name)).get(name));

                    ArrayList<TM_TypeEvaluator> typeList =
                        this.recordMapType.get(name);

315                 int i = 0;
                    for(TM_ValueExpr ve : applicationExpr.
                        value_expr_list().getList()) {
                        if(ve.
                            getTypeEvaluatorForTypeEvaluation
                            () == null) {
                            if(ve instanceof
                                Make_TM_ListExpr) {
                                ((Make_TM_ListExpr) ve).list_expr
                                    ().setTypeEvaluatorForTypeEvaluation
                                    (typeList.get(i));
320                         }
                            else {
                                ve.setTypeEvaluatorForTypeEvaluation
                                    (typeList.get(i));
                            }
                            //System.out.println("Setting ve
                                : " + ve + " : " + typeList.
                                get(i));
325                     }
                        i++;
                    }
                }
            }
330         else if(ooExternalMethodList.contains(id.getText
                ())) {
                //System.out.println("Must be an application
                     of an extra inserted getMethod: " + id.
                    getText());
                applicationExpr.
                    setTypeEvaluatorForTypeEvaluation(new
                    TM_TypeEvaluator_TM_Destructor(new
                    TM_Destructor(id)));
                applicationExpr.value_expr().
                    setTypeEvaluatorForTypeEvaluation(new
                    TM_TypeEvaluator_TM_Destructor(new
                    TM_Destructor(id)));

335             //applicationExpr.
                    setTypeEvaluatorForTypeEvaluation(new
```

```
                    Destructor(id));
            }
        }
        else if(map.get(id.getText()) != null && this.
            functionMapType.get(id.getText()) == null) {
            /*Record variant*/
            //System.out.println("Must be a record variant
                or destructor: " + id + " to: " + (map.get(id
                .getText())).get(id.getText()));

            applicationExpr.
                setTypeEvaluatorForTypeEvaluation(map.get(id.
                getText()).get(id.getText()));

            if(this.recordMapType.get(id.getText()) != null)
                {
                ArrayList<TM_TypeEvaluator> typeList = new
                    ArrayList<TM_TypeEvaluator>();
                typeList = this.recordMapType.get(id.getText
                    ());

                int i = 0;
                for(TM_ValueExpr ve : applicationExpr.
                    value_expr_list().getList()) {
                    if(ve.getTypeEvaluatorForTypeEvaluation
                        () == null) {
                        if(ve instanceof Make_TM_ListExpr) {
                            ((Make_TM_ListExpr) ve).list_expr
                                ().setTypeEvaluatorForTypeEvaluation
                                (typeList.get(i));
                        }
                        else {
                            ve.setTypeEvaluatorForTypeEvaluation
                                (typeList.get(i));
                        }
                        //System.out.println("Setting ve
                            : " + ve + " : " + typeList.get(i
                            ));
                    }
                i++;
                }
            }
        }
        else {
            //System.out.println("Must be an application of
                a function: " + id + " to: " + (map.get(id.
                getText())).get(id.getText()));
            //System.out.println("FunctionResultType: " +
                functionResultType);
```

```
                ArrayList<TM_TypeEvaluator> typeList = new
                    ArrayList<TM_TypeEvaluator>();
                typeList = this.functionMapType.get(id.getText()
                    );

                applicationExpr.
                    setTypeEvaluatorForTypeEvaluation(
                    functionResultType.get(id.getText()));

                if(debug && applicationExpr.getParent()
                    instanceof TM_ValueInfixExpr)
                    System.out.println(id.getText() + " to: " +
                                       functionResultType.get(id
                                           .getText()));

                int i = 0;
                for(TM_ValueExpr ve : applicationExpr.
                    value_expr_list().getList()) {
                    if(ve.getTypeEvaluatorForTypeEvaluation()
                        == null) {
                        if(ve instanceof Make_TM_ListExpr) {
                            ((Make_TM_ListExpr) ve).list_expr().
                                setTypeEvaluatorForTypeEvaluation
                                (typeList.get(i));
                        }
                        else {
                            ve.setTypeEvaluatorForTypeEvaluation
                                (typeList.get(i));
                        }
                        //System.out.println("Setting ve: " + ve
                            + " : " + typeList.get(i));
                    }
                    i++;
                }
            }
            if(applicationExpr.getTypeEvaluatorForTypeEvaluation
                () == null && this.debug) {
                //System.out.println("Cannot set type of
                    application expression: " + applicationExpr.
                    value_expr());
            }
        }
    }

    public void visitMake_TM_ListExpr(Make_TM_ListExpr
        make_ListExpr) {
        make_ListExpr.list_expr().accept(this);
        if(make_ListExpr.list_expr().
```

```
            getTypeEvaluatorForTypeEvaluation() != null)
             make_ListExpr.setTypeEvaluatorForTypeEvaluation(
                 make_ListExpr.list_expr().
                 getTypeEvaluatorForTypeEvaluation());
400     }

        public void visitTM_EnumeratedListExpr(TM_EnumeratedListExpr
            enumeratedListExpr) {
            for(TM_ValueExpr ve : enumeratedListExpr.value_expr_list
                ().getList()) {
                ve.accept(this);
405         }
            if(!enumeratedListExpr.value_expr_list().getList().
                isEmpty()) {
                //System.out.println("Setting type via hd of list!")
                    ;
                //System.out.println("ParentMap: " + parentMap);
                if(enumeratedListExpr.value_expr_list().hd().
410                 getTypeEvaluatorForTypeEvaluation() instanceof
                        TM_TypeEvaluator_TM_Constructor) {
                    enumeratedListExpr.
                        setTypeEvaluatorForTypeEvaluation(
                        new TM_TypeEvaluator_TM_TypeExpr(
                            new TM_TypeExprList(new
                                TM_FiniteListTypeExpr(
                                new TM_TypeName(new TM_Id(parentMap
                                    .get(((
                                    TM_TypeEvaluator_TM_Constructor)
                                     enumeratedListExpr.
                                    value_expr_list().hd().
415                                 getTypeEvaluatorForTypeEvaluation()).
                                        constructor().id().getText()))))
                    ))));
                    //System.out.println("hd setting type to: " +
                        enumeratedListExpr.
                        getTypeEvaluatorForTypeEvaluation());
                }
                else if(enumeratedListExpr.value_expr_list().hd().
                    getTypeEvaluatorForTypeEvaluation() instanceof
                    TM_TypeEvaluator_TM_TypeExpr) {
420                 enumeratedListExpr.
                        setTypeEvaluatorForTypeEvaluation(
                        new TM_TypeEvaluator_TM_TypeExpr(
                            new TM_TypeExprList(new
                                TM_FiniteListTypeExpr(
                                ((TM_TypeEvaluator_TM_TypeExpr)
                                    enumeratedListExpr.value_expr_list()
                                    .hd().
                                    getTypeEvaluatorForTypeEvaluation())
```

```
                                .type_expr()
                        ))));
425             }
                else {
                    //System.out.println("Type of enumerated list
                        could not be set: " + enumeratedListExpr.
                        value_expr_list().hd());
                    //System.exit(1);
                }
430         }
            else {
                //System.out.println("Setting type via call to
                    parent!");
                enumeratedListExpr.setTypeEvaluatorForTypeEvaluation
                    (enumeratedListExpr.getParent().
                    getTypeEvaluatorForTypeEvaluation());
                //if(enumeratedListExpr.
                    getTypeEvaluatorForTypeEvaluation() == null &&
                    enumeratedListExpr.getParent().getParent().
                    getParent() != null) {
435             //enumeratedListExpr.
                    setTypeEvaluatorForTypeEvaluation(
                    enumeratedListExpr.getParent().getParent().
                    getParent().getTypeEvaluatorForTypeEvaluation());
                //}
            }
            if(enumeratedListExpr.getTypeEvaluatorForTypeEvaluation
                () == null && this.debug) {
                //System.out.println("Cannot set type of
                    enumeratedList expr: " + enumeratedListExpr.
                    getParent().getParent().getParent());
440             //System.out.println("Type of 3xparent: " +
                    enumeratedListExpr.getParent().getParent().
                    getParent().getTypeEvaluatorForTypeEvaluation());
                //System.out.println("Type of 2xparent: " +
                    enumeratedListExpr.getParent().getParent().
                    getTypeEvaluatorForTypeEvaluation());
            }
        }

445     public void visitMake_TM_IfExpr(Make_TM_IfExpr makeIfExpr) {
            makeIfExpr.if_expr().accept(this);
        }

        public void visitTM_IfExpr(TM_IfExpr ifExpr) {
450         ifExpr.condition().accept(this);
            ifExpr.if_case().accept(this);
            for(TM_Elsif elsif : ifExpr.elsif_list().getList()) {
                elsif.accept(this);
```

```
            }
455         ifExpr.else_case().accept(this);

            //Get type of first branch and use it as type of whole
                expr
            if(ifExpr.if_case().getTypeEvaluatorForTypeEvaluation()
                != null) {
                ifExpr.setTypeEvaluatorForTypeEvaluation(ifExpr.
                    if_case().getTypeEvaluatorForTypeEvaluation());
460             ifExpr.else_case().setTypeEvaluatorForTypeEvaluation
                    (ifExpr.if_case().
                    getTypeEvaluatorForTypeEvaluation());
                //System.out.println("Setting type of IfExpr via if
                    branch to: " + ifExpr.
                    getTypeEvaluatorForTypeEvaluation() );
            }
            else if(ifExpr.else_case().
                getTypeEvaluatorForTypeEvaluation() != null) {
                ifExpr.setTypeEvaluatorForTypeEvaluation(ifExpr.
                    else_case().getTypeEvaluatorForTypeEvaluation());
465             ifExpr.if_case().setTypeEvaluatorForTypeEvaluation(
                    ifExpr.else_case().
                    getTypeEvaluatorForTypeEvaluation());
                //System.out.println("Setting type of IfExpr via
                    else branch: " + ifExpr.if_case().
                    getTypeEvaluatorForTypeEvaluation());
            }
            else {
                //System.out.println("Type of if expression not set
                    !!!");
470         }

        }

        public void visitTM_Elsif(TM_Elsif elsif) {
475         elsif.condition().accept(this);
            elsif.elsif_case().accept(this);
        }

        public void visitTM_CaseExpr(TM_CaseExpr caseExpr) {
480         caseExpr.condition().accept(this);
            for(TM_CaseBranch cb : caseExpr.case_branch_list().
                getList()) {
                cb.accept(this);
            }
            for(TM_CaseBranch cb : caseExpr.case_branch_list().
                getList()) {
485             if(cb.getTypeEvaluatorForTypeEvaluation() != null) {
                    caseExpr.setTypeEvaluatorForTypeEvaluation(cb.
```

```
                    getTypeEvaluatorForTypeEvaluation());
                break;
            }
        }
        if(caseExpr.getTypeEvaluatorForTypeEvaluation() != null)
            {
            for(TM_CaseBranch cb : caseExpr.case_branch_list().
                getList()) {
                if(cb.getTypeEvaluatorForTypeEvaluation() ==
                    null)
                    cb.setTypeEvaluatorForTypeEvaluation(
                        caseExpr.
                        getTypeEvaluatorForTypeEvaluation());
            }
        }
        else {
            if(caseExpr.getParent().
                getTypeEvaluatorForTypeEvaluation() != null) {
                caseExpr.setTypeEvaluatorForTypeEvaluation(
                    caseExpr.getParent().
                    getTypeEvaluatorForTypeEvaluation());
            }
            else {
                //if(this.debug)
                //    System.out.println("Cannot set type of
                    case expr: " + caseExpr.condition());
            }
        }

    }

    public void visitTM_CaseBranch(TM_CaseBranch caseBranch) {
        caseBranch.pattern().accept(this);
        caseBranch.value_expr().accept(this);

        if(caseBranch.value_expr().
            getTypeEvaluatorForTypeEvaluation() != null) {
            caseBranch.setTypeEvaluatorForTypeEvaluation(
                caseBranch.value_expr().
                getTypeEvaluatorForTypeEvaluation());
        }
        else {
            //System.out.println("Cannot evaluate expression.
                Try setting via parent: " + caseBranch.getPattern
                ());
            //Try set via parent
            if(caseBranch.getParent() != null) {
                if(caseBranch.getParent().
                    getTypeEvaluatorForTypeEvaluation() != null)
```

```
                    {
520                 caseBranch.setTypeEvaluatorForTypeEvaluation
                        (caseBranch.getParent().
                        getTypeEvaluatorForTypeEvaluation());
                    if(caseBranch.value_expr() instanceof
                        Make_TM_ListExpr) {
                        //if(this.debug)
                        //    System.out.println("Setting via
                            parent: " + caseBranch + " to: " +
                            caseBranch.getParent().
                            getTypeEvaluatorForTypeEvaluation());

525                     ((Make_TM_ListExpr) caseBranch.
                            value_expr()).list_expr().
                            setTypeEvaluatorForTypeEvaluation(
                            caseBranch.getParent().
                            getTypeEvaluatorForTypeEvaluation());
                    }
                    else
                        caseBranch.value_expr().
                            setTypeEvaluatorForTypeEvaluation(
                            caseBranch.getParent().
                            getTypeEvaluatorForTypeEvaluation());
                }
530             else {
                    //System.out.println("Parent has no type");
                }
            }
            else {
535             //System.out.println("Parent not set!!!");
            }
        }
    }

540 public void visitTM_ValueLiteralPattern(
        TM_ValueLiteralPattern pattern) {
        pattern.value_literal().accept(this);
    }

    public void visitTM_RecordPattern(TM_RecordPattern pattern)
        {
545     pattern.value_or_variable_name().accept(this);

        ArrayList<TM_TypeEvaluator> tl = recordMapType.get(
            pattern.value_or_variable_name().id().getText());
        ArrayList<String> il = recordMapId.get(pattern.
            value_or_variable_name().id().getText());
        /*
550     int i = 0;
```

```
        for(TM_Pattern p : pattern.inner_pattern_list().getList
            ()) {
            p.accept(this);
            ((TM_NamePattern) p).
                setTypeEvaluatorForTypeEvaluation(tl.get(i));
            currentMap.put(((TM_NamePattern)p).id().getText(),
                tl.get(i));
            ((TM_NamePattern) p).setValueInitializer(new TM_Id(
                il.get(i)));
            i++;
        }
        */
        for(int i = 0; i < pattern.inner_pattern_list().len(); i
            ++) {
            TM_NamePattern p = (TM_NamePattern) pattern.
                inner_pattern_list().getList().get(i);
            currentMap.put(p.id().getText(), tl.get(i));
            TM_NamePattern np = new TM_NamePattern(p.id(), new
                Make_TM_Id(new TM_Id(il.get(i))));
            np.setTypeEvaluatorForTypeEvaluation(tl.get(i));
            pattern.inner_pattern_list().getList().set(i, np);
        }
    }

    public void visitTM_NamePattern(TM_NamePattern pattern) {
        pattern.id().accept(this);
    }

    public void visitTM_WildcardPattern(TM_WildcardPattern
        pattern) {}

    public void visitTM_ValueInfixExpr(TM_ValueInfixExpr
        valueInfixExpr) {
        valueInfixExpr.left().accept(this);
        valueInfixExpr.op().accept(this);
        valueInfixExpr.right().accept(this);

        if(valueInfixExpr.left().
            getTypeEvaluatorForTypeEvaluation() != null) {
            valueInfixExpr.setTypeEvaluatorForTypeEvaluation(
                valueInfixExpr.left().
                getTypeEvaluatorForTypeEvaluation());
            if(valueInfixExpr.right().
                getTypeEvaluatorForTypeEvaluation() == null) {
                if(valueInfixExpr.right() instanceof
                    Make_TM_ListExpr) {
                    ((Make_TM_ListExpr) valueInfixExpr.right()).
                        list_expr().
                        setTypeEvaluatorForTypeEvaluation(
```

```
                        valueInfixExpr.left().
                            getTypeEvaluatorForTypeEvaluation());
                    valueInfixExpr.right().
                        setTypeEvaluatorForTypeEvaluation(
                        valueInfixExpr.left().
                        getTypeEvaluatorForTypeEvaluation());
                }
                else {
                    valueInfixExpr.right().
                        setTypeEvaluatorForTypeEvaluation(
                        valueInfixExpr.left().
                        getTypeEvaluatorForTypeEvaluation());
                }
            }
            else {

            }
        }
        else if(valueInfixExpr.right().
            getTypeEvaluatorForTypeEvaluation() != null) {
            valueInfixExpr.setTypeEvaluatorForTypeEvaluation(
                valueInfixExpr.right().
                getTypeEvaluatorForTypeEvaluation());
            //if(valueInfixExpr.left() == null)
            valueInfixExpr.left().
                setTypeEvaluatorForTypeEvaluation(valueInfixExpr.
                right().getTypeEvaluatorForTypeEvaluation());
        }
        else {
            //System.out.println("Type of ValueInfixExpr not set
                : " + valueInfixExpr);
            //System.exit(1);
        }
        //System.out.println("ValueInfixExpr.
            getTypeEvaluatorForTypeEvaluation(): " +
            valueInfixExpr.getTypeEvaluatorForTypeEvaluation());
    }

    public void visitTM_ValuePrefixExpr(TM_ValuePrefixExpr
        valuePrefixExpr) {
        valuePrefixExpr.op().accept(this);
        valuePrefixExpr.operand().accept(this);

        TM_PrefixOperator op = valuePrefixExpr.op();
        if(op instanceof TM_RSL_HD) {
            TM_ValueExpr ve = valuePrefixExpr.operand();
            if(ve instanceof Make_TM_ValueOrVariableName) {
                TM_ValueOrVariableName vovn = ((
                    Make_TM_ValueOrVariableName) ve).
```

```
                value_or_variable_name();
615             //HOPEFULLY THE TYPE OF THE VARIABLE OR VALUE IS
                    A FINITELISTTYPEEXPR
                //IF NOT THERE MUST HAVE BEEN AN ERROR. WE SET
                    THE TYPE OF THE EVALUATOR TO BE WHAT IS
                    INSIDE THE LIST


                //System.out.println("CurrentMap: " + currentMap
                    );
620             //System.out.println("VOVN: " + (vovn.id().
                    getText()));


                valuePrefixExpr.
                    setTypeEvaluatorForTypeEvaluation(
                     new TM_TypeEvaluator_TM_TypeExpr(
625                      ((TM_FiniteListTypeExpr)(((
                             TM_TypeExprList) (((
                             TM_TypeEvaluator_TM_TypeExpr)
                             currentMap.get(vovn.id().getText())).
                             type_expr()))).type_expr_list()).
                             type_expr()
                     )
                     );

                //valuePrefixExpr.
                    setTypeEvaluatorForTypeEvaluation(currentMap.
                    get(vovn.id().getText()));
630         }
            else if(ve instanceof Make_TM_ListExpr) {
                Make_TM_ListExpr le = (Make_TM_ListExpr)
                    valuePrefixExpr.operand();
                if(le.list_expr() instanceof
                    TM_EnumeratedListExpr) {
                     TM_EnumeratedListExpr ele = (
                         TM_EnumeratedListExpr) le.list_expr();
635                  valuePrefixExpr.
                         setTypeEvaluatorForTypeEvaluation(ele.
                         getTypeEvaluatorForTypeEvaluation());
                }
            }
        }
        else if(op instanceof TM_RSL_TL) {
640         TM_ValueExpr ve = valuePrefixExpr.operand();
            if(ve instanceof Make_TM_ValueOrVariableName) {
                //System.out.println("Operand of prefix_expr is
                    a variable!");
                TM_ValueOrVariableName vovn = ((
```

```
                    Make_TM_ValueOrVariableName) ve).
                    value_or_variable_name();
                //HOPEFULLY THE TYPE OF THE VARIABLE OR VALUE IS
                    A FINITELISTTYPEEXPR
645             //IF NOT THERE MUST HAVE BEEN AN ERROR. WE SET
                    THE TYPE OF THE EVALUATOR TO BE WHAT IS
                    INSIDE THE LIST
                valuePrefixExpr.
                    setTypeEvaluatorForTypeEvaluation(currentMap.
                    get(vovn.id().getText()));
            }
            else if(ve instanceof Make_TM_ListExpr) {
                //System.out.println("Operand of prefix_expr is
                    an enumerated list expr!");
650             Make_TM_ListExpr le = (Make_TM_ListExpr)
                    valuePrefixExpr.operand();;
                if(le.list_expr() instanceof
                    TM_EnumeratedListExpr) {
                    TM_EnumeratedListExpr ele = (
                        TM_EnumeratedListExpr) le.list_expr();
                    if(ele.getTypeEvaluatorForTypeEvaluation()
                        instanceof TM_TypeEvaluator_TM_TypeExpr)
                        valuePrefixExpr.
                            setTypeEvaluatorForTypeEvaluation(new
                             TM_TypeEvaluator_TM_TypeExpr(new
                            TM_TypeExprList(new
                            TM_FiniteListTypeExpr(((
                            TM_TypeEvaluator_TM_TypeExpr)ele.
                            getTypeEvaluatorForTypeEvaluation()).
                            type_expr())))));
655                 else if(ele.
                        getTypeEvaluatorForTypeEvaluation()
                        instanceof
                        TM_TypeEvaluator_TM_Constructor)
                        valuePrefixExpr.
                            setTypeEvaluatorForTypeEvaluation(new
                             TM_TypeEvaluator_TM_TypeExpr(new
                            TM_TypeExprList(new
                            TM_FiniteListTypeExpr(new TM_TypeName
                            (((TM_TypeEvaluator_TM_Constructor)
                            ele.getTypeEvaluatorForTypeEvaluation
                            ()).constructor().id())))));
                }
            }
        }
660 }

    public void visitTM_ParenthesizedExpr(TM_ParenthesizedExpr
        parenthesizedExpr) {
```

```
        parenthesizedExpr.parenthesized_expr().accept(this);
        if(parenthesizedExpr.parenthesized_expr().
            getTypeEvaluatorForTypeEvaluation() != null)
665          parenthesizedExpr.setTypeEvaluatorForTypeEvaluation(
                 parenthesizedExpr.parenthesized_expr().
                 getTypeEvaluatorForTypeEvaluation());
        else if(parenthesizedExpr.getParent() != null) {
             if(parenthesizedExpr.getParent().
                 getTypeEvaluatorForTypeEvaluation() != null)
                  parenthesizedExpr.
                      setTypeEvaluatorForTypeEvaluation(
                      parenthesizedExpr.getParent().
                      getTypeEvaluatorForTypeEvaluation());
        }
670     }

        public void visitTM_RSL_EQUAL(TM_RSL_EQUAL rsl_equal) {}

        public void visitTM_RSL_PLUS(TM_RSL_PLUS rsl_plus) {}
675
        public void visitTM_RSL_HD(TM_RSL_HD rsl_hd) {}

        public void visitTM_RSL_TL(TM_RSL_TL rsl_tl) {}

680     public void visitMake_TM_ValueOrVariableName(
            Make_TM_ValueOrVariableName valueOrVariableName) {
             valueOrVariableName.value_or_variable_name().accept(this
                 );
             valueOrVariableName.setTypeEvaluatorForTypeEvaluation(
                 valueOrVariableName.value_or_variable_name().
                 getTypeEvaluatorForTypeEvaluation());
        }

685     public void visitTM_ValueOrVariableName(
            TM_ValueOrVariableName valueOrVariableName) {
             valueOrVariableName.id().accept(this);

             if(map != null && map.get(valueOrVariableName.id().
                 getText()) != null)
                  valueOrVariableName.
                      setTypeEvaluatorForTypeEvaluation((map.get(
                      valueOrVariableName.id().getText())).get(
                      valueOrVariableName.id().getText()));
690          else if(currentMap != null && currentMap.get(
                 valueOrVariableName.id().getText()) != null)
                  valueOrVariableName.
                      setTypeEvaluatorForTypeEvaluation(currentMap.get(
                      valueOrVariableName.id().getText()));
        }
```

```
    public void visitMake_TM_ValueLiteral(Make_TM_ValueLiteral
        valueLiteral) {
695      valueLiteral.value_literal().accept(this);
         valueLiteral.setTypeEvaluatorForTypeEvaluation(
             valueLiteral.value_literal().
             getTypeEvaluatorForTypeEvaluation());
    }

    public void visitTM_ValueLiteralInteger(
        TM_ValueLiteralInteger valueLiteralInteger) {
700      valueLiteralInteger.setTypeEvaluatorForTypeEvaluation(
             new TM_TypeEvaluator_TM_TypeExpr(new TM_TypeLiteral(
             new TM_RSL_INT())));
    }

    public void visitTM_ValueLiteralReal(TM_ValueLiteralReal
        valueLiteralReal) {
         valueLiteralReal.setTypeEvaluatorForTypeEvaluation(new
             TM_TypeEvaluator_TM_TypeExpr(new TM_TypeLiteral(new
             TM_RSL_REAL())));
705  }

    public void visitTM_ValueLiteralBool(TM_ValueLiteralBool
        valueLiteralBool) {
         valueLiteralBool.setTypeEvaluatorForTypeEvaluation(new
             TM_TypeEvaluator_TM_TypeExpr(new TM_TypeLiteral(new
             TM_RSL_BOOL())));
    }
710
    public void visitTM_ValueLiteralChar(TM_ValueLiteralChar
        valueLiteralChar) {
         valueLiteralChar.setTypeEvaluatorForTypeEvaluation(new
             TM_TypeEvaluator_TM_TypeExpr(new TM_TypeLiteral(new
             TM_RSL_CHAR())));
    }

715 public void visitTM_ValueLiteralText(TM_ValueLiteralText
        valueLiteralText) {
         valueLiteralText.setTypeEvaluatorForTypeEvaluation(new
             TM_TypeEvaluator_TM_TypeExpr(new TM_TypeLiteral(new
             TM_RSL_TEXT())));
    }


720  /*Common*/
    public void visitTM_Binding(TM_Binding binding) {}

    public void visitTM_Id(TM_Id id) {
```

```
            if(currentMap != null && !currentMap.containsKey(id.
                getText())) {
725             currentMap.put(id.getText(), null);
            }
        }
    }
```

Appendix G

Test Code

G.1 Test of library classes

G.1.1 RSLListDefaultTest.java

```
import translator.rsllib.*;

public class RSLListDefaultTest {
    public static void main(String[] args) {
        RSLListDefault<Integer> l1 = new RSLListDefault<Integer
            >(new Integer[]{6,7,8});
        System.out.println("l1: " + l1);

        RSLListDefault<Integer> l2 = new RSLListDefault<Integer
            >(1,5);
        System.out.println("l2: " + l2);

        RSLListDefault<Integer> l3 = l1.listComp(
           new RSLExpression<Integer>(){public Integer action(
               Integer i){return 2*i;}},
           new Testable<Integer>(){public boolean test(Integer i
               ){return (i % 2 == 0);}});
        System.out.println("l3: " + l2);
        System.out.println("l1: " + l1 +
                           " to ensure the original list where
                               not changed");

        RSLList<Integer> l4 = l1.concat(l2);
        System.out.println("l4 = l1.concatenate(l2): " + l4);
        System.out.println("l1: " + l1 +
                           " to ensure the original list where
                               not changed");
        System.out.println("l2: " + l2 +
```

```
                        " to ensure the original list where
                            not changed");

        int i = l1.hd();
        System.out.println("i = l1.hd(): " + i);
        System.out.println("l1: " + l1 +
                        " to ensure the original list where
                            not changed");

        RSLList<Integer> l5 = l2.tl();
        System.out.println("l5 = l2.tl(): " + l5);
        System.out.println("l2: " + l2 +
                        " to ensure the original list where
                            not changed");

        int j = l5.len();
        System.out.println("j = l5.len(): " + j);
        System.out.println("l5: " + l5 +
                        " to ensure the original list where
                            not changed");

        RSLList<Integer> l6 = l1.concat(l2);
        System.out.println("l4 = l1.concatenate(l2): " + l4);
        System.out.println("l1: " + l1 +
                        " to ensure the original list where
                            not changed");

        RSLSet<Integer> s1 = new RSLListDefault<Integer>(new
            Integer[]{2,2,4,4,4,5,5}).elems();
        System.out.println("s1 = new RSLListDefault<Integer>(" +
                        "new Integer[]{2,2,4,4,4,5,5}).elems
                            (): \n" + s1);


        RSLSet<Integer> s2 = new RSLListDefault<Integer>(new
            Integer[]{2,2,4,4,4,5,5}).inds();
        System.out.println("s2 = new RSLListDefault<Integer>(" +
                        "new Integer[]{2,2,4,4,4,5,5}).inds()
                            : \n" + s2);

        int k = l2.get(3);
        System.out.println("k = l2.get(3): " + k);

        System.out.println("(new RSLListDefault<Integer>(new
            Integer[]{7,8,9}).set(2,10): " +
                        (new RSLListDefault<Integer>(new
                            Integer[]{7,8,9}).set(2,10)));

```

```
    }
}
```

G.1.2 RSLSetDefaultTest.java

```
import translator.rsllib.*;

public class RSLSetDefaultTest {
    public static void main(String[] args) {
        System.out.println("Starting main method");
        RSLSet<Integer> s1 = new RSLSetDefault<Integer>(1);
        System.out.println("s1: " + s1.toString());
        RSLSet<Double> s2 = new RSLSetDefault<Double>(new Double
            []{1.2, 2.3});
        System.out.println("s2: " + s2.toString());
        RSLSet<Integer> s3 = new RSLSetDefault<Integer>(3);
        System.out.println("s3: " + s3.toString());
        System.out.println("s3.union(s1): " + s3.union(s1));
        System.out.println("s3.union(s1).isIn(4): " + s3.union(
            s1).isIn(4));
        System.out.println("s3.union(s1).isNotIn(4): " + s3.
            union(s1).isNotIn(4));
        System.out.println("s1: " + s1.toString());
        System.out.println("s3: " + s3.toString());

        RSLSet<Integer> s4 = new RSLSetDefault<Integer>(new
            Integer[]{1,2,3,4,5});
        RSLSet<Integer> s5 = new RSLSetDefault<Integer>(new
            Integer[]{1,3,5});
        System.out.println("s4.intersect(s5): " + s4.intersect(
            s5));
        System.out.println("s4: " + s4.toString());
        System.out.println("s5: " + s5.toString());

        RSLSet<Integer> s6 = new RSLSetDefault<Integer>(new
            Integer[]{1,3,5});
        RSLSet<Integer> s7 = new RSLSetDefault<Integer>(new
            Integer[]{1,3,5});
        System.out.println("s6.superSet(s7): " + s6.superSet(s7)
            );
        System.out.println("s6: " + s6.toString());
        System.out.println("s7: " + s7.toString());

        RSLSet<Integer> s8 = new RSLSetDefault<Integer>(new
            Integer[]{1,3,5});
        RSLSet<Integer> s9 = new RSLSetDefault<Integer>(new
```

```
            Integer[]{1,3,5});
        System.out.println("s8.properSuperSet(s9): " + s8.
            properSuperSet(s9));
        System.out.println("s8: " + s8.toString());
        System.out.println("s9: " + s9.toString());

        RSLSet<Integer> s10 = new RSLSetDefault<Integer>(new
            Integer[]{1,2,3,4,5});
        RSLSet<Integer> s11 = new RSLSetDefault<Integer>(new
            Integer[]{1,3,5});
        System.out.println("s10 .difference(s11): " + s10.
            difference(s11));
        System.out.println("s10: " + s10.toString());
        System.out.println("s11: " + s11.toString());

        System.out.println("Ending main method");
    }
}
```

G.1.3 RSLMapDefaultTest.java

```
import translator.rsllib.*;

public class RSLMapDefaultTest {
    public static void main(String[] args) {
        System.out.println("Starting main method");
        RSLMap<Integer, String> m1 = new RSLMapDefault<Integer,
            String>(new Integer[]{1,2,3}, new String[]{"one", "
            two", "three"});
        System.out.println("m1: " + m1);
        System.out.println("m1.dom(): " + m1.dom());
        System.out.println("m1.rng(): " + m1.rng());
        System.out.println("m1: " + m1);
        System.out.println("m1.restrictTo(new RSLSetDefault<
            Integer>(1)): " + m1.restrictTo(new RSLSetDefault<
            Integer>(1)));
        System.out.println("m1: " + m1);
        System.out.println("m1.restrictBy(new RSLSetDefault<
            Integer>(new Integer[]{2,3})): " + m1.restrictTo(new
            RSLSetDefault<Integer>(1)));
        System.out.println("m1: " + m1);
        System.out.println("m1.override(new RSLMapDefault<
            Integer, String>(1, \"four\")):\n " + m1.override(new
             RSLMapDefault<Integer,String>(1,"four")));
        System.out.println("m1: " + m1);

        RSLMap<String, Double> m2 = new RSLMapDefault<String,
            Double>(new String[]{"two","three"}, new Double[]{2.0
            d, 3.0d});
```

```
        System.out.println("m2: " + m2);
        System.out.println("m2.compose(m1): " + m2.compose(m1));
        System.out.println("m1: " + m1);
        System.out.println("m2: " + m2);
        System.out.println("Ending main method");
    }
}
```

Appendix H

Test Results

H.1 Library classes

H.1.1 RSLListDefaultTest

```
l1: <.6,7,8.>
l2: <.1,2,3,4,5.>
l3: <.1,2,3,4,5.>
l1: <.6,7,8.> to ensure the original list where not changed
l4 = l1.concatenate(l2): <.6,7,8,1,2,3,4,5.>
l1: <.6,7,8.> to ensure the original list where not changed
l2: <.1,2,3,4,5.> to ensure the original list where not changed
i = l1.hd(): 6
l1: <.6,7,8.> to ensure the original list where not changed
l5 = l2.tl(): <.2,3,4,5.>
l2: <.1,2,3,4,5.> to ensure the original list where not changed
j = l5.len(): 4
l5: <.2,3,4,5.> to ensure the original list where not changed
l4 = l1.concatenate(l2): <.6,7,8,1,2,3,4,5.>
l1: <.6,7,8.> to ensure the original list where not changed
s1 = new RSLListDefault<Integer>(new Integer[]{2,2,4,4,4,5,5}).elems():
{2,4,5}
s2 = new RSLListDefault<Integer>(new Integer[]{2,2,4,4,4,5,5}).inds():
{2,4,6,1,3,7,5}
k = l2.get(3): 3
(new RSLListDefault<Integer>(new Integer[]{7,8,9}).set(2,10): <.7,10,9.>
```

H.1.2 RSLLSetDefaultTest

```
Starting main method
s1: [1]
s2: [2.3, 1.2]
s3: [3]
s3.union(s1): [1, 3]
s3.union(s1).isIn(4): false
s3.union(s1).isNotIn(4): true
s1: [1]
s3: [3]
s4.intersect(s5): [1, 3, 5]
s4: [2, 4, 1, 3, 5]
s5: [1, 3, 5]
s6.superSet(s7): true
s6: [1, 3, 5]
s7: [1, 3, 5]
s8.properSuperSet(s9): false
s8: [1, 3, 5]
s9: [1, 3, 5]
s10 .difference(s11): [2, 4]
s10: [2, 4, 1, 3, 5]
s11: [1, 3, 5]
Ending main method
```

H.1.3 RSLMapDefaultTest

```
Starting main method
m1: {2=two, 1=one, 3=three}
m1.dom(): [2, 1, 3]
m1.rng(): [one, two, three]
m1: {2=two, 1=one, 3=three}
m1.restrictTo(new RSLSetDefault<Integer>(1)): {1=one}
m1: {2=two, 1=one, 3=three}
m1.restrictBy(new RSLSetDefault<Integer>(new Integer[]{2,3})): {1=one}
m1: {2=two, 1=one, 3=three}
m1.override(new RSLMapDefault<Integer, String>(1, "four")):
 {2=two, 1=four, 3=three}
m1: {2=two, 1=one, 3=three}
m2: {two=2.0, three=3.0}
m2.compose(m1): {2=2.0, 3=3.0}
```

```
15  m1: {2=two, 1=one, 3=three}
    m2: {two=2.0, three=3.0}
    Ending main method
```

Appendix I

Examples

I.1 LISTSUM

I.1.1 LISTSUM.rsl

```
scheme LISTSUM =
  class
    value
      listMult : Int* × Int → Int*
      listMult(il, i) ≡
        if il = ⟨⟩ then
          ⟨⟩
        else
          ⟨ (hd il ∗ i) ⟩ ⌢ listMult(tl il, i)
        end,
      listSum : Int* → Int
      listSum(il) ≡ if il = ⟨⟩ then 0 else hd il + listSum(tl il) end,
      listDiv : Int* × Int → Int*
      listDiv(il, i) ≡
        if il = ⟨⟩ then
          ⟨⟩
        else
          ⟨ (hd il / i) ⟩ ⌢ listDiv(tl il, i)
        end
    test_case
      [ t1 ] listSum(⟨1,2,3⟩),
      [ t2 ] listSum(⟨5⟩),
      [ t3 ] listSum(⟨⟩),
      [ t4 ] listMult(⟨1,2,3⟩, 2),
```

[t5] listMult(⟨5⟩, 2),
[t6] listMult(⟨⟩, 2),
[t7] listSum(listMult(⟨1,2,3⟩, 2)),
[t8] listMult(⟨2,4,6⟩, 2),
[t9] listDiv(⟨2,4,6⟩, 2)
end

I.1.2 LISTSUM.java

```
//26 October 2004 16:00:13 CEST
//NO PACKAGE DECLARATION
import java.util.*;
import translator.rsllib.*;

public class LISTSUM {
//METHOD DECLARATION
public static RSLList<Integer> listMult(RSLList<Integer> il, int
    i) {
RSLList<Integer> _v0 = null;
if(il.equals(new RSLListDefault<Integer>())){
RSLList<Integer> _v1 = null;
_v1 = new RSLListDefault<Integer>();
_v0 = _v1;
}
else{
RSLList<Integer> _v1 = null;
_v1 = new RSLListDefault<Integer>(new Integer[]{(il.hd() * i)}).
   concat(listMult(il.tl(), i));
_v0 = _v1;
}
return _v0;
}
//METHOD DECLARATION
public static int listSum(RSLList<Integer> il) {
int _v0 = 0;
if(il.equals(new RSLListDefault<Integer>())){
int _v1 = 0;
_v1 = 0;
_v0 = _v1;
}
else{
int _v1 = 0;
_v1 = il.hd() + listSum(il.tl());
_v0 = _v1;
}
return _v0;
```

```
}
//METHOD DECLARATION
public static RSLList<Integer> listDiv(RSLList<Integer> il, int
    i) {
RSLList<Integer> _v0 = null;
if(il.equals(new RSLListDefault<Integer>())){
RSLList<Integer> _v1 = null;
_v1 = new RSLListDefault<Integer>();
_v0 = _v1;
}
else{
RSLList<Integer> _v1 = null;
_v1 = new RSLListDefault<Integer>(new Integer[]{(il.hd() / i)}).
    concat(listDiv(il.tl(), i));
_v0 = _v1;
}
return _v0;
}
//METHOD DECLARATION
public static void main(String[] args) {
System.out.println("[t1]: " + listSum(new RSLListDefault<Integer
    >(new Integer[]{1, 2, 3})));
System.out.println("[t2]: " + listSum(new RSLListDefault<Integer
    >(new Integer[]{5})));
System.out.println("[t3]: " + listSum(new RSLListDefault<Integer
    >()));
System.out.println("[t4]: " + listMult(new RSLListDefault<
    Integer>(new Integer[]{1, 2, 3}), 2));
System.out.println("[t5]: " + listMult(new RSLListDefault<
    Integer>(new Integer[]{5}), 2));
System.out.println("[t6]: " + listMult(new RSLListDefault<
    Integer>(), 2));
System.out.println("[t7]: " + listSum(listMult(new
    RSLListDefault<Integer>(new Integer[]{1, 2, 3}), 2)));
System.out.println("[t8]: " + listMult(new RSLListDefault<
    Integer>(new Integer[]{2, 4, 6}), 2));
System.out.println("[t9]: " + listDiv(new RSLListDefault<Integer
    >(new Integer[]{2, 4, 6}), 2));
}
}
```

I.1.3 Result of Execution

```
[t1]: 6
[t2]: 5
[t3]: 0
[t4]: <.2,4,6.>
[t5]: <.10.>
```

```
[t6]: <..>
[t7]: 12
[t8]: <.4,8,12.>
[t9]: <.1,2,3.>
```

I.2 LIST

I.2.1 LIST.rsl

scheme LIST =
class
  type
    MyList == Empty | Add(head : **Int**, tail : MyList)
  value
    addOne: MyList → MyList
    addOne(ml) ≡
      case ml **of**
        Empty → Empty,
        Add(i, m1) → Add(i + 1, addOne(m1))
      end
  test_case
    [t1] Add(1, Add(2, Add(3, Empty))),
    [t2] addOne(Add(1, Add(2, Add(3, Empty))))
end

I.2.2 List.java

```
//13 October 2004 11:02:34 CEST
//NO PACKAGE DECLARATION
import java.util.*;
import translator.rsllib.*;

public class LIST {
//METHOD DECLARATION
public static MyList addOne(MyList ml) {
MyList _v0 = null;
if(ml instanceof Empty){
MyList _v1 = null;
_v1 = new Empty();
_v0 = _v1;
}
else{
```

```
if(ml instanceof Add){
int i = ((Add)ml).head();
MyList m1 = ((Add)ml).tail();
MyList _v1 = null;
_v1 = new Add(i + 1, addOne(m1));
_v0 = _v1;
}
}
return _v0;
}
//METHOD DECLARATION
public static void main(String[] args) {
System.out.println("[t1]: " + new Add(1, new Add(2, new Add(3,
    new Empty()))));
System.out.println("[t2]: " + addOne(new Add(1, new Add(2, new
    Add(3, new Empty())))));
}
}
```

I.2.3 MyList.java

```
//13 October 2004 11:02:34 CEST
//NO PACKAGE DECLARATION
import java.util.*;
import translator.rsllib.*;

public abstract class MyList {
//METHOD DECLARATION
public abstract boolean equals(Object o) ;
//METHOD DECLARATION
public abstract String toString() ;
}
```

I.2.4 Empty.java

```
//13 October 2004 11:02:34 CEST
//NO PACKAGE DECLARATION
import java.util.*;
import translator.rsllib.*;

public class Empty extends MyList {
//METHOD DECLARATION
public boolean equals(Object o) {
return o instanceof Empty;
}
//METHOD DECLARATION
public String toString() {
return "Empty";
}
```

```
}
```

I.2.5 Add.java

```
//13 October 2004 11:02:34 CEST
//NO PACKAGE DECLARATION
import java.util.*;
import translator.rsllib.*;

public class Add extends MyList {
//FIELD DECLARATION
private int _v0;
//FIELD DECLARATION
private MyList _v1;
public Add(int _v0, MyList _v1) {
this._v0 = _v0;
this._v1 = _v1;
}
//METHOD DECLARATION
public boolean equals(Object o) {
if(!(this.head() == ((Add)o).head())){
return false;
}
if(this.tail().equals(((Add)o).tail())){
return false;
}
return true;
}
//METHOD DECLARATION
public String toString() {
return "Add(" + this.head() + ", " + this.tail().toString() + ")
    ";
}
//METHOD DECLARATION
public int head() {
return _v0;
}
//METHOD DECLARATION
public MyList tail() {
return _v1;
}
}
```

I.2.6 Result of Execution

```
[t1]: Add(1, Add(2, Add(3, Empty)))
[t2]: Add(2, Add(3, Add(4, Empty)))
```

I.3 APPLICATION

I.3.1 APPLICATION.rsl

scheme APPLICATION =
class
  type
    My_ShortRecordDefinition :: getInt : **Int** getReal : **Real**
  value
    function : **Unit** → **Int**
    function() ≡ 5
  test_case
    [t1] mk_My_ShortRecordDefinition(5, 7.0),
    [t2] function(),
    [t3] getInt(mk_My_ShortRecordDefinition(2, 3.0))
end

I.3.2 APPLICATION.java

```
//13 October 2004 10:41:04 CEST
//NO PACKAGE DECLARATION
import java.util.*;
import translator.rsllib.*;

public class APPLICATION {
//METHOD DECLARATION
public static int function() {
int _v0 = 0;
_v0 = 5;
return _v0;
}
//METHOD DECLARATION
public static void main(String[] args) {
System.out.println("[t1]: " + new My_ShortRecordDefinition
    (5, 7.0));
System.out.println("[t2]: " + function());
System.out.println("[t3]: " + (new My_ShortRecordDefinition
    (2, 3.0)).getInt());
}
}
```

I.3.3 MyShortRecordDefinition.java

```
//13 October 2004 10:41:04 CEST
//NO PACKAGE DECLARATION
import java.util.*;
import translator.rsllib.*;

public class My_ShortRecordDefinition {
//FIELD DECLARATION
private int _v0;
//FIELD DECLARATION
private double _v1;
public My_ShortRecordDefinition(int _v0, double _v1) {
this._v0 = _v0;
this._v1 = _v1;
}
//METHOD DECLARATION
public boolean equals(Object o) {
if(!(this.getInt() == ((My_ShortRecordDefinition)o).getInt())){
return false;
}
if(!(this.getReal() == ((My_ShortRecordDefinition)o).getReal()))
    {
return false;
}
return true;
}
//METHOD DECLARATION
public String toString() {
return "My_ShortRecordDefinition(" + this.getInt() + ", " + this
    .getReal() + ")";
}
//METHOD DECLARATION
public int getInt() {
return _v0;
}
//METHOD DECLARATION
public double getReal() {
return _v1;
}
}
```

I.3.4 Result of Execution

```
[t1]: My_ShortRecordDefinition(5, 7.0)
[t2]: 5
[t3]: 2
```