Technical Reports in Computer and Information Science
Report number 2011:10

Towards Modelica 4 Meta-Programming and Language Modeling with MetaModelica 2.0

by

Peter Fritzson, Adrian Pop, and Martin Sjölund
{peter.fritzson, adrian.pop, martin.sjolund}@liu.se

Draft, March 18, 2011

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Technical reports in Computer and Information Science
ISSN: 1654-7233
Year: 2011
Report no. 10

Available online at Linköping University Electronic Press
http://www.ep.liu.se/PubList/Default.aspx?SeriesID=2550

Towards Modelica 4 Meta-Programming and Language Modeling with MetaModelica 2.0

by

Peter Fritzson, Adrian Pop, and Martin Sjölund

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Draft, March 18, 2011

Available online at Linköping University Electronic Press
http://www.ep.liu.se/PubList/Default.aspx?SeriesID=2550

Abstract

This report gives a language definition and tutorial on how to model languages using MetaModelica 2.0 – an extension of Modelica 3.2 designed for efficient language modeling. Starting from an extremely simple language, a series of small languages are modeled by gradually adding features. Both interpretive and translational language semantics are modeled. Exercises with solutions are given.

The approach of allowing the modeling language to model language semantics in principle allows the definition of language semantics in libraries, which could be used to reverse the current trend of model compilers becoming very large and complex.

MetaModelica 2.0 builds on MetaModelica 1.0 which was the first Modelica language version that supports language modeling, and has been in extensive use since 2005, primarily in the development of the OpenModelica compiler.

The following version of MetaModelica, called MetaModelica 2.0, is described in this report. It is easier to use since it also supports the standard Modelica 3 language features as well as additional modeling features for expressiveness and conciseness. It is implemented within the OpenModelica compiler itself. This means that the OpenModelica compiler supporting MetaModelica 2.0 is bootstrapped, i.e., it compiles itself.

This work is strongly connected to the Modelica 4 effort announced by Modelica Association in September 2010, which includes moving language functionality into library packages to achieve more extensible and modular Modelica model compilers. The MetaModelica language features contribute to realizing that goal. The language features have been proven in large-scale usage in the packages within the OpenModelica compiler. However, much work still remains in improving the modularity and interface properties that are expected by library packages.

Table of Contents

Preface 13
Acknowledgements 14

Chapter 1 Extensible Tools, Language Modeling, and Tool Generation 15

1.1 Language Modeling for Extensible Tool Functionality 15
1.2 Generation of Language Processing Tools from Specifications 16
1.3 Using MetaModelica for Modeling of Programming Languages 16
1.4 Compiler Generation 17
1.5 Interpreter Generation 18
1.6 Bootstrapping 19

Chapter 2 Expression Evaluators and Interpreters in MetaModelica 21

2.1 The Exp1 Expression Language 21
2.1.1 Concrete Syntax 21
2.1.2 Abstract Syntax of Exp1 with Union Types 22
2.1.3 The uniontype Construct 22
2.1.4 Semantics of Exp1 23
2.1.4.1 Match-Expressions in MetaModelica 23
2.1.4.2 Evaluation of the Exp1 Language 24
2.1.4.3 Using Named Pattern Matching for Exp1 26
2.2 Exp2 – Using Parameterized Abstract Syntax 27
2.2.1 Parameterized Abstract Syntax of Exp1 28
2.2.2 Parameterized Abstract Syntax of Exp2 28
2.2.3 Semantics of Exp2 29
2.2.3.1 Tuples in MetaModelica 29
2.2.3.2 The Exp2 Evaluator 29
2.3 Recursion and Failure in MetaModelica 31
2.3.1 Short Introduction to Declarative Programming in MetaModelica 31
2.3.2 Handling Failure 31
2.4 The Assignments Language – Introducing Environments 32
2.4.1 Environments 33
2.4.2 Concrete Syntax of the Assignments Language 33
2.4.3 Abstract Syntax of the Assignments Language 34
2.4.4 Semantics of the Assignments Language 36
2.4.4.1 Semantics of Lookup in Environments 36
2.4.4.2 Updating and Extending Environments at Lookup 37
2.4.4.3 Evaluation Semantics 39
2.5 PAM – Introducing Control Structures and I/O 40
2.5.1 Examples of PAM Programs 40
2.5.2 Concrete Syntax of PAM 41
2.5.3 Abstract Syntax of PAM 43
2.5.4 Semantics of PAM 44
2.5.4.1 Expression Evaluation 44
2.5.4.2 Arithmetic and Relational Operators 45

2.5.4.3 Statement Evaluation....46
2.5.4.4 Auxiliary Functions....48
2.5.4.5 Repeated Statement Evaluation....49
2.5.4.6 Error Handling....49
2.5.4.7 Stream I/O Primitives....49
2.5.4.8 Environment Lookup and Update....50
2.6 AssignTwoType – Introducing Typing....50
2.6.1 Concrete Syntax of AssignTwoType....50
2.6.2 Abstract Syntax....51
2.6.3 Semantics of AssignTwoType....52
2.6.3.1 Expression Evaluation....52
2.6.3.2 Type Lattice and Least Upper Bound....54
2.6.3.3 Binary and Unary Operators....55
2.6.3.4 Functions for Lookup and Environment Update....56
2.7 A Modular Specification of the PAMDECL Language....57
2.8 Summary....57
2.9 Exercises....58

Chapter 3 Translational Semantics....59

3.1 Translating PAM to Machine Code....60
3.1.1 A Target Assembly Language....60
3.1.2 A Translated PAM Example Program....61
3.1.3 Abstract Syntax for Machine Code Intermediate Form....62
3.1.4 Concrete Syntax of PAM....62
3.1.5 Abstract Syntax of PAM....63
3.1.6 Translational Semantics of PAM....63
3.1.6.1 Arithmetic Expression Translation....63
3.1.6.2 Translation of Comparison Expressions....66
3.1.6.3 Statement Translation....68
3.1.6.4 Emission of Textual Assembly Code....73
3.1.6.5 Translate a PAM Program and Emit Assembly Code....76
3.2 The Semantics of Mcode....76
3.3 Translational Semantics for Symbolic Differentiation....76
3.4 Summary....78
3.5 Exercises....79

Chapter 4 Getting Started – Practical Details....81

4.1 Activating MetaModelica....81
4.2 Path and Locations of Needed Files....81
4.2.1 Windows Dependencies....81
4.2.2 Linux and Mac OSX Dependencies....82
4.2.3 Downloading the Examples....82
4.3 Examples of Using MetaModelica from OMNotebook....82
4.3.1 Union Type Number Example....83
4.3.1.1 Small Number Exercise....83
4.3.2 Exp1Real Calculator in the Notebook....84
4.3.3 List Data Structures....85
4.3.4 Tuples....86
4.3.5 Option Types....86
4.4 The MDT Eclipse Plugin....87
4.5 The Exp1 Calculator Again....87
4.5.1 Running the Exp1 Calculator....87
4.5.2 Building the Exp1 Calculator....88
4.5.2.1 Source Files to be Provided....88
4.5.2.2 Generated Source Files....88
4.5.2.3 Library File(s)....88
4.5.2.4 Makefile for Building the Exp1 Calculator....89

4.5.3 Source Files for the Exp1 Calculator....90
4.5.3.1 Lexical Syntax: lexer.l....90
4.5.3.2 Grammar: parser.y....91
4.5.3.3 Semantics: Exp1.mo....93
4.5.3.4 Calling the calculator....93
4.5.4 Library Files....94
4.5.4.1 meta_modelica.h....94
4.6 An Evaluator for PAMDECL....95
4.6.1 Running the PAMDECL Evaluator....95
4.6.2 Building the PAMDECL Evaluator....95
4.6.3 Calling C from MetaModelica....95

Chapter 5 Introductory Overview of MetaModelica 2.0....97

5.1 Miscellaneous Features....97
5.1.1 MetaModelica Keywords....97
5.1.2 Predefined Types and Type Constructors....97
5.1.3 Builtin MetaModelica Functions for Basic Boxed Data Types....98
5.1.5 Built-in Special Operators and Functions....98
5.1.6 Arithmetic Exceptions....99
5.1.7 MetaModelica Constructs to Possibly be Deprecated....99
5.2 Comparison Between MetaModelica 1.0 and 2.0....99
5.2.1 Changes to MetaModelica Builtin Types....100
5.3 MRArray – Globally Mutable Ragged Arrays vs Standard Arrays....101
5.3.1 Modelica Arrays, Mutability, and Time-Dependent Simulation....101
5.3.2 The MRArray Data Type....101

Chapter 6 Collection Types and Operations....103

6.1 Parameterized Data Types....103
6.2 Sequence Type and { } Constructor....103
6.3 Lists....104
6.3.1 List Types....105
6.3.2 List Literals....105
6.4 Arrays....106
6.5 Conversions between Arrays, MRArrays, Lists, and Sequences....107
6.6 For-loops, List and Array Comprehensions with Guards....108
6.6.1 Guarded Patterns in for-loops....110

Chapter 7 Union Types, Options, and Pattern Matching....111

7.1 Structured Data Types....111
7.1.1 Tuples....111
7.1.2 Union Types for Records, Trees, and Graphs....111
7.1.2.1 Parameterized Union Types and Record Types....112
7.1.3 Option Types....113
7.2 Pattern-Matching and Match-Expressions....113
7.2.1 Syntactic Structure of Match-Expressions....114
7.2.1.1 Match-Expressions with Local Algorithm Sections....115
7.2.2 Match-Statements....115
7.2.3 Evaluation of Match-Expressions and Match-Statements....117
7.2.4 Forms of Patterns....117
7.2.4.1 Guarded Patterns....118
7.2.4.2 Patterns with the as Binding Operator....118
7.2.4.3 Positional Argument Constructor Call Patterns....119
7.2.4.4 Named Argument Constructor Call Patterns....120
7.2.5 Compact Matching and Named Field Access via Dot Notation....121
7.3 Usage of Patterns....122
7.3.1 Examples....122
7.3.2 Patterns in Left-hand-sides of Equations or Statements....122

7.3.3 Constraint Equations 122
7.3.4 Constructing Values 123
7.4 Forms of Equations in Match Cases 123
7.4.1 Logically Overlapping Patterns 123
7.4.2 Default Case in Match-Expressions 124

Chapter 8 Failures, Exceptions and Error Reporting 125

8.1 Function Failure Versus Boolean Negation 125
8.2 Failure and Retry in Functions with matchcontinue 126
8.3 Error Reporting Location Information 126

Chapter 9 Type Inference and Implicit Type Conversion 129

9.1 Types in Declaration Equations and Declaration Assignments 129
9.1.1 Potential Problems Using Type Inference 129
9.2 Types Values of Polymorphic Type Variables in Functions 130
9.3 Inferring Types of as-Bound Variables 130
9.4 Inferring Types of Input Variables to Pattern Matching 131
9.5 Implicit Type Conversion 131
9.5.1 The { } Constructor and Implicit Conversions to Arrays and Lists 132
9.5.1.1 Implicit Conversion from Sequence to Array 132
9.5.1.2 Implicit Conversion from Sequence to List 133
9.5.1.1 Implicit Conversion from Sequence to MRArray 133

Chapter 10 Functions 135

10.1 Function Declaration 135
10.2 Builtin Functions 136
10.3 Pure and Impure Functions 136
10.4 Failure and Retry in Functions 137
10.5 Argument Passing and Result Values 138
10.5.1 Tuple Arguments and Results 138
10.5.2 Passing Functions as Arguments to Functions 138
10.6 Variables and Types in Functions 139
10.6.1 Type Variables and Parameterized Types in Functions 139
10.6.2 Local Variables in Match-Expressions in Functions 140
10.6.3 Function Failure Versus Boolean Negation 140
10.6.4 Last Call Optimization – Tail Recursion Removal 140
10.6.5 The Method of Accumulating Parameters for Collecting Results 141
10.6.6 Using Side Effects in Specifications 142
10.7 Examples of Higher-Order Programming with Functions 144
10.7.1 Reducing a List to a Scalar Using listReduce 144
10.7.2 Mapping a Function Over a List Using listMap 144

Chapter 11 Packages and Import 147

11.1 Compact Syntax for Multiple Definition Import 147
11.2 Importing Union Type Record Constructors 148
11.2.1 Implicit Import of Union Type Record Constructors 148

Chapter 12 Text Template Language 149

12.1 Definition of Text Template Language 149
12.2 Design Principles 150
12.3 Preliminaries 150
12.3.1 Predefined Text Data Type 150
12.3.2 Differences and Similarities Between the Text Template Language and MetaModelica 151
12.4 Template Text-with-hole Constructors and Template Holes 151
12.5 Template Expressions 152
12.5.1 Automatic Conversion to String Data 152
12.5.2 Bound Variable Reference, name or $'name' 153

12.5.3 Verbatim String Constants 153
12.5.4 Parentheses 153
12.5.5 Reduction Expressions 153
12.5.6 Conditional Expressions 154
12.5.7 List Constructor {} 154
12.6 Template Function Declaration and Call 155
12.6.1 Template Function Declaration 155
12.6.2 Declaring External Imported MetaModelica Functions 155
12.6.3 Template Function Call 155
12.6.4 Call of Imported External MetaModelica or C Functions 156
12.6.4.1 Using return value natively with its original type 156
12.6.4.2 Call with return value converted to string 156
12.6.4.3 Call with empty return value 157
12.6.5 Declaring Reference (buffer) Formal Parameters in a Template Function 157
12.6.6 Using Reference (buffer) Formal Parameters in a Template Function 157
12.6.7 Allowed Side-Effects in Template Functions 158
12.7 Match Expressions and the Idea of Pattern Matching 158
12.7.1 The MetaModelica uniontype Construct 158
1.1.1 Match Expressions and Pattern Matchin3 159
12.7.1.1 Simple Pattern Matching 159
12.7.1.2 Using Pattern Matching in Template Functions 160
12.7.2 Pattern Expressions in General 161
12.7.3 Record Constructor Pattern Expressions 161
12.7.4 Pattern Expression Variable Binding Using as 162
12.7.5 Implicit Opening of Record Constructor Scopes in Patterns 162
12.8 Iterator Expressions 162
12.8.1 Iterator Expressions with Iteration Index Values 164
12.9 Let Expressions with Name Bindings and Text Buffers 164
12.9.1 let Binding of Local Named Text Values 164
12.9.2 let Binding of Buffer Variables and their Use 165
12.9.3 Appending a String to a buffer Variable 165
12.9.4 Reference (buffer) Formal Parameters in Template Functions 166
12.10 Formatting, Separator, and Indentation Options 166
12.10.1 Indentation controlling options 166
12.10.2 Multi-Value Formatting Options 166
12.10.3 Options 167
12.11 Interface Packages 168
12.11.1 Interface Package Import into a Template File 170
12.12 Template File Import into Another Template File 171
12.13 Template Error Handling 171

Appendix A MetaModelica Grammar 173

A.1 Class and Main Grammar Elements 173
A.2 Extends 175
A.3 Modification 176
A.4 Equations 176
A.5 Expressions 179
A.6 MetaModelica Extensions 181

Appendix B Predefined MetaModelica Operators and Functions 183

B.1 Precedence of Predefined Operators 183
B.2 Short Descriptions of Builtin Functions and Operators 183
B.3 Interface to the Standard MetaModelica Package 185
B.3.1 Predefined Types and Type Constructors 185
B.3.2 Boolean Operations 185
B.3.3 Integer Operations 186

B.3.4 Real Number Operations 188
B.3.5 String Character Conversion Operations 191
B.3.6 String Operations 191
B.3.7 List Operations 193
B.3.8 MRArray Operations 193
B.3.9 Miscellaneous Operations 195

Appendix C Complete Small Language Specifications 197

C.1 The Complete Interpretive Semantics for PAM 197
C.2 Complete PAMDECL Interpretive Specification 203
C.2.1 PAMDECL Main Package 203
C.2.2 PAMDECL ScanParse 203
C.3 PAMDECL Absyn 204
C.3.1 PAMDECL Env 205
C.3.2 PAMDECL Eval 207
C.3.3 PAMDECL lexer.l 215
C.3.4 PAMDECL parser.y 217
C.3.5 PAMDECL Makefile 220
C.4 The Complete PAM Translational Specification 221
C.4.1 PAMTRANS Absyn.mo 221
C.4.2 PAMTRANS Trans.mo 222
C.4.3 PAMTRANS Mcode.mo 227
C.4.4 PAMTRANS Emit.mo 229
C.4.5 PAMTRANS lexer.l 233
C.4.6 PAMTRANS parser.y 235
C.4.7 PAMTRANS Main.mo 238
C.4.8 PAMTRANS Parse.mo 239
C.4.9 PAMTRANS Makefile 239

Appendix D Exercises 241

D.1 Exercises – Introduction and Interpretive Semantics 241
D.1.1 Exercise 01_experiment – Types, Functions, Constants, Printing Values 241
D.1.2 Exercise 02a_Exp1 – Adding New Features to a Small Language 243
D.1.3 Exercise 02b_Exp2 – Adding New Features to a Small Language 245
D.1.4 Exercise 03 Symbolic Derivative – Differentiating an Expression Using Symbolic Manipulation 246
D.1.5 Exercise 04 Assignment – Printing AST and Environments 252
D.1.6 Exercise 04a_AssignTwoType – Adding a New Type to a Language 256
D.1.7 Exercise 04b_ModAssigntwotype – Modularized Specification 262
D.2 Exercises – Translational Semantics 263
D.2.1 Exercise 09_pamtrans – Small Translational Semantics 263
D.2.2 Exercise 10_Petrol – Large Translational Semantics 263
D.3 Exercises – Advanced 263
D.3.1 Exercise 05_advanced – Polymorphic Types and Higher Order Functions 263

Appendix E Solutions to Exercises 265

E.1 Solution 01_experiment – Types, Functions, Constants, Printing Values 265
E.2 Solution 02a_Exp1 – Adding New Features to a Small Language 269
E.3 Solution 03 Symbolic Derivative – Differentating an Expression Using Symbolic Manipulation 270
E.4 Solution 04 Assignment – Printing AST and Environments 276
E.5 Solution 05a AssignTwoType – Adding a New Type to a Language 281
E.6 Solution 05b ModAssignTwoType – Adding a New Type to a Language 287
E.7 Solution 06 Advanced – Polymorphic Types and Higher Order Functions 288
E.8 Solution 07_pam – A small Language 292
E.9 Solution 08_pamdecl – Pam with Declarations 292
E.10 Solution 09_pamtrans – Small Translational Semantics 292

E.11 Solution 10_Petrol – Large Translational Semantics 292
References 293
Index 297

Preface

The work on MetaModelica has its roots in our early work on executable specification languages for defining the semantics of programming languages and generating efficient compilers from such specifications. This started during the late 1980s with Peter Fritzson's and his students' work on attribute grammars and denotational semantics based tools. During the beginning of the 1990s the focus was changed into support for executable language specifications in the popular Natural Semantics/Structured Operational Semantics formalism, 1995 resulting in the RML tool as the PhD thesis work by Mikael Pettersson. This tool and formalism was first used for the specification of several languages: both imperative, functional, and object-oriented (Java 1.2). During 1997/98 the first formal specification of a subset of Modelica was developed, which influenced the early Modelica specification. This specification grew over time, and eventually developed into the OpenModelica open source effort.

At the same time, we and others made the observation that since user requirements on the usage of models grow over time, and the scope of modeling domains increase, the demands on the Modelica modeling language and corresponding tools increase. This has caused the Modelica language and model compilers to become increasingly large and complex.

One approach to manage this increasing complexity used by several functional languages is to define a number of language features in libraries rather than in the compiler itself.

Why not apply this idea to the Modelica language? However, the language modeling features needed, e.g. found in RML and similar languages, were missing in standard Modelica. Therefore, during 2004-2005 we designed and implemented a language extension to Modelica called MetaModelica. The first implementation included the development of a MetaModelica 1.0 compiler frontend, but still used the RML core compiler and code generator. This implementation had the advantage of rather quickly making the MetaModelica 1.0 language available for use. Moreover, extensive work on the modeling environment (Eclipse plug-in, debugger) was needed to make it effective for large-scale use by the developers.

The MetaModelica 1.0 language has been in extensive use during 2005-2011, primarily for development of the OpenModelica compiler. However, the MetaModelica 1.0 language has the drawback of not supporting many features in the standard Modelica language, and lacking some language constructs that would make the specifications more readable and concise.

The next version of MetaModelica, called MetaModelica 2.0, is described in this report. It is easier to use since it supports the standard Modelica 3 language features as well as additional modeling features for expressiveness and conciseness. It is implemented within the OpenModelica compiler itself and is not dependent on the old RML compiler kernel. This means that the OpenModelica compiler supporting MetaModelica 2.0 is bootstrapped, i.e., it compiles itself. MetaModelica 2.0 became operational during spring 2011.

This work is strongly connected to the Modelica 4 effort announced by Modelica Association in September 2010, which includes moving language functionality into library packages to achieve more extensible and modular Modelica model compilers. The MetaModelica 2.0 language features contribute to realizing that goal. The language features have been proven in large-scale usage in the packages within the OpenModelica compiler. However, much work still remains in improving the modularity and interface properties that are expected by library packages.

Linköping, Sweden, March 2011

Peter Fritzson, Adrian Pop, and Martin Sjölund

Acknowledgements

Pavol Privitzer is the main implementor and designer of the OpenModelica text template language described in Chapter 12. He has also contributed to the text of that chapter.

Many users of MetaModelica have given constructive comments and feedback over the years. We would especially like to mention Peter Aronsson in this respect.

Vinnova through the OPENPROD ITEA2 project and SSF (Swedish Strategic Research Foundation) through the Proviking HiPo project have provided partial financial support for this work.

Chapter 1

Extensible Tools, Language Modeling, and Tool Generation

In this chapter we briefly discuss the concept of extensibility of modeling, analysis, and simulation tools, and how this can be realized by extending the modeling language to also specify language properties and symbolic transformations.

The rest of this book is organized as follows. Chapter 1 to Chapter 3 give an introduction to the topic of generating compilers and interpreters from MetaModelica specifications, giving a tutorial introduction through a series of small languages. Chapter 4 describes practical issues how to get started, whereas Chapter 5 to Chapter 11 describe the MetaModelica language extensions. Chapter 12 describes a domain specific text template language strongly related to MetaModelica. It is used to specify code generation to any text representation, e.g., target languages such as C, C#, XML.

For the reader already familiar with MetaModelica 1.0, a comparison and summary of new features in MetaModelica 2.0 is available in Section 5.2. Some of the new features in MetaModelica 2.0 were not completely implemented at the time of publishing this report. A description of the implementation status is available in the document www.ida.liu.se/~petfr/MetaModelica2.0Status.pdf.

1.1 Language Modeling for Extensible Tool Functionality

Traditionally, a model compiler performs the task of translating a model into executable code, which then is executed during simulation of the model. Thus, the symbolic translation step is followed by an execution step, a simulation, which often involves large-scale numeric computations.

However, as requirements on the usage of models grow, and the scope of modeling domains increases, the demands on the modeling language and corresponding tools increase. This causes the model compiler to become large and complex.

Moreover, the modeling community needs not only tools for simulation but also languages and tools to create, query, manipulate, and compose equation-based models. Additional examples are optimization of models, parallelization of models, checking and configuration of models.

If all this functionality is added to the model compiler, it tends to become large and complex.

An alternative idea is to add features to the modeling language such that for example a model package can contain model analysis and translation features that therefore are not required in the model compiler. An example is a PDE discretization scheme that could be expressed in the modeling language itself as part of a PDE package instead of being added internally to the model compiler.

In this text we will primarily describe language constructs and examples of their usage in specifying languages and tools for different processing tasks.

1.2 Generation of Language Processing Tools from Specifications

The implementation of language processing tools such as compilers and interpreters for non-trivial programming languages is a complex and error prone process, if done by hand. Therefore, formalisms and generator tools have been developed that allow automatic generation of compilers and interpreters from formal specifications. This offers two major advantages:

- High-level descriptions of language properties, rather than detailed programming of the translation process.
- High degree of correctness of generated implementations.

The high level specifications are typically more concise and easier to read than a detailed implementation in some traditional low-level programming language. The declarative and modular specification of language properties rather than detailed operational description of the translation process, make it much easier to verify the logical consistency of language constructs and to detect omissions and errors. This is virtually impossible for a traditional implementation, which often requires time consuming debugging and testing to obtain a compiler of acceptable quality. By using automatic compiler generation tools, correct compilers can be produced in a much shorter time than otherwise possible. This, however, requires the availability of generator tools of high quality which can produce compiler components with a performance comparable to hand-written ones.

1.3 Using MetaModelica for Modeling of Programming Languages

The Modelica specification and modeling language was originally developed as an object-oriented declarative equation-based specification formalism for mathematical modeling of complex systems, in particular physical systems.

However, it turns out that with some minor extensions, the Modelica language is well suited for another modeling task, namely modeling of the semantics, i.e., the meaning, of programming language constructs. Since modeling of programming languages is often known as meta-modeling, we use the name MetaModelica for this slightly extended Modelica. The semantics of a language construct can usually be modeled in terms of combinations of more primitive builtin constructs. One example of primitive builtin operations are the integer arithmetic operators. These primitives are combined using inference and pattern-matching mechanisms in the specification language.

Well-known language specification formalisms such as Natural Semantics (Despeyroux 1984; Despeyroux 1988; Pettersson 1995; Fritzson 1996; Fritzson and Kågedal 1998) and Structured Operational Semantics (Plotkin 1981; Mosses 2004) are also declarative equation-based formalisms. These fit well into the style of the MetaModelica specification language, which explains why Modelica with some minor extensions is well-suited as a language specification formalism. However, only an extended subset of Modelica called MetaModelica is needed for language specification since many parts of the language designed for physical system modeling are not used at all, or very little, for the language specification task.

This text introduces the use of MetaModelica for programming language specification, in a style reminiscent of Natural or Structured Operational Semantics, but using Modelica's properties for enhanced readability and structure.

Another great benefit of using and extending Modelica in this direction is that the language becomes suitable for meta-programming and meta-modeling. This means that Modelica can be used for transformation of models and programs, including transforming and combining Modelica models into other Modelica models.

However, the main emphasis in the rest of Chapter 1 to Chapter 3 is on the topic of writing specifications in MetaModelica for the generation of compilers and interpreters.

1.4 Compiler Generation

The process of compiler generation is the automatic production of a compiler from formal specifications of source language, target language, and various intermediate formalisms and transformations. This is depicted in Figure 1-1, which also shows some examples of compiler generation tools and formalisms for the different phases of a typical compiler. Classical tools such as scanner generators (e.g. Lex) and parser generators (e.g. Yacc) were first developed in the 1970:s. Many similar generation tools for producing scanners and parsers exist.

However, the semantic analysis and intermediate code generation phase is still often hand-coded, although attribute grammar based tools have been available for practical usage for quite some time. Even though attribute grammars are easy to use for certain aspects of language specifications, they are less convenient when used for many other language aspects. Specifications tend to become long and involve many details and dependencies on external functions, rather than clearly expressing high level properties. Denotational Semantics is a formalism that provides more abstraction power, but is considered hard to use by most practitioners, and has problems with modularity of specifications and efficiency of produced implementations. We will not further discuss the matter of different specification formalisms, and refer the reader to other literature, e.g. (Louden 2003) and (Pierce2002).

Semantic aspects of language translation include tasks such as type checking/type inference, symbol table handling, and generation of intermediate code. If automatic generation of translator modules for semantic tasks should become as common as generation of parsers from BNF grammars, we need a specification formalism that is both easy to use and that provides a high degree of abstraction power for expressing language translation and analysis tasks. The MetaModelica formalism fulfils these requirements. We have therefore chosen this formalism for semantics specification in this text.

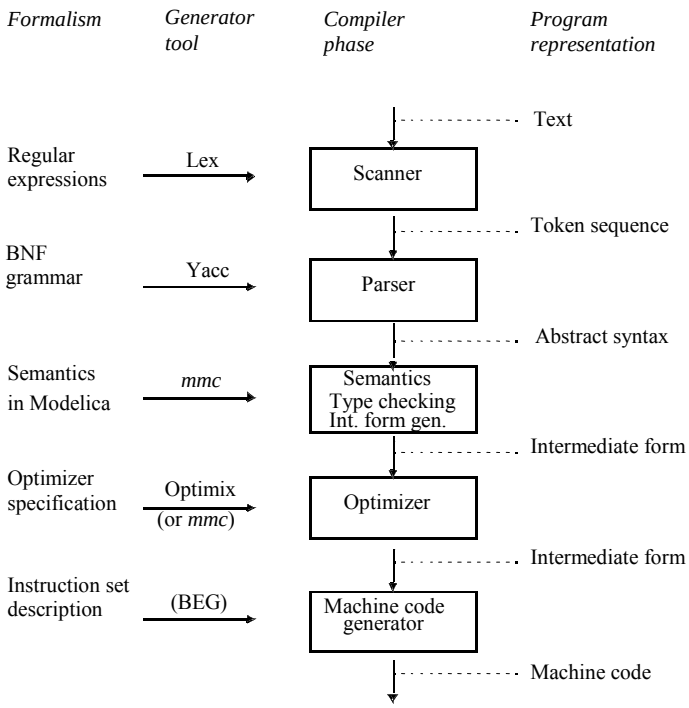

Figure 1-1. Generation of implementations of compiler phases from different formalisms. MetaModelica is used to specify the semantics module, which is generated using the OpenModelica compiler.

The second necessary requirement for widespread practical use of automatic generation of semantics parts of language implementations is that the generated result needs to be roughly as efficient as hand-written implementations., a generator tool, `omc` (OpenModelica Compiler), that produces highly efficient implementations in C—roughly of the same efficiency as hand-written ones, and a debugger for debugging specifications. MetaModelica also enables modularity of specification through a module

system with packages, and interfaceability to other tools since the generated modules in C can be readily combined with other frontend or backend modules.

The later phases of a compiler, such as optimization of the intermediate code and generation of machine code are also often hand-coded, although code generator generators such as BEG (Landwehr, Jansohn, Goos 1982; Emmelmann, Schröer, Landwehr 1989), IBURG (Fraser and Hansen 1995), and their use (Andersson and Fritzson 1996) have been developed during the late 1980s and early 1990:s. A product version of BEG is available in the CoSy compiler generation toolbox (ACE 2011) which also includes global register allocation and instruction scheduling. A university version is described in (Alt 1997).

The optimization phase of compilers is generally hand coded, although some prototypes of optimizer generators have appeared. For example, an optimizer generator tool called Optimix (Assmann 2000) has influenced the tools in the CoSy (ACE 2011) compiler generation system.

MetaModelica can also be used for these other phases of compilers, such as optimization of intermediate code and final code generation. Intermediate code optimization works rather well since this is usually a combination of analysis and transformation that can take advantage of patterns, tree transformation expressions, and other features of the MetaModelica language.

Regarding final machine code generation modules of most compilers – these are probably best produced by specialized tools such as BEG, which use specific algorithms such as dynamic programming for "optimal" instruction selection, and graph coloring for register allocation. However, in this book we only present a few very simple examples of final code generation, and essentially no examples of advanced code optimization.

1.5 Interpreter Generation

The case of generating an interpreter from formal specifications can be regarded as a simplified special case of compiler generation. Although some systems interpret text directly (e.g. command interpreters such as the Unix C shell), most systems first perform lexical and syntactic analysis to convert the program into some intermediate form, which is much more efficient to interpret than the textual representation. Type checking and other checking is usually done at run-time, either because this is required by the language definition (as for many interpreted languages such as LISP, Postscript, Smalltalk, etc.), or to minimize the delay until execution is started.

The semantic specification of a programming language intended as input for the generation of an interpreter if usually slightly different in style compared to a specification intended for compiler generation. Ideally, they would be exactly the same, and there exist techniques such as partial evaluation (Jones, Gomard, Sestoft, 1993; Wikipedia 2011), that sometimes can produce compilers also from specifications of interpreters.

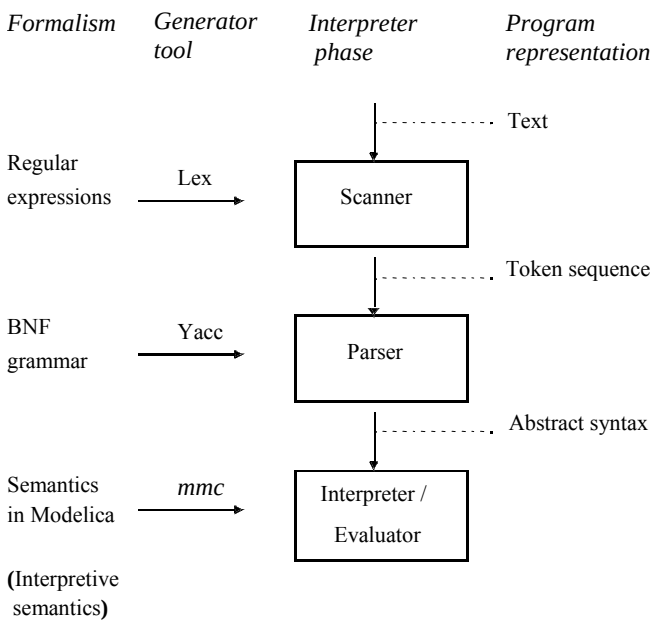

Figure 1-2. Generation of a typical interpreter. The program text is converted into an abstract syntax representation, which is then evaluated by an interpreter generated by the OpenModelica Compiler system. Alternatively, some other intermediate representation such as postfix code can be produced, which is subsequently interpreted.

In practice, an interpretive style specification often expresses the meaning of a language construct by invoking a combination of well-defined primitives in the specification language. A compilation oriented specification, however, usually defines the meaning of language constructs by specifying a translation to an equivalent combination of well-defined constructs in some target language. In this text we will show examples of both interpretive and translation-oriented specifications.

1.6 Bootstrapping

The term bootstrapping means that a language is used to define itself, implying that the compiler for that language is used to compile itself. Figuratively speaking, you lift yourself in your own boot straps, which of course is impossible.

What is possible, however, is to write an executable language specification in a subset or approximation of the same language, implement that subset in some way, and compile it. Then we have a full specification language compiler implemented in a subset. Finally, the specification is updated/refactored to take advantage of the full specification language. The end product is a fully bootstrapped language and compiler, compiling itself.

In the case of MetaModelica, we started by developing a Natural Semantics style specification in RML (Pettersson 1995; Pettersson 1999; Fritzson 1996) for a subset of Modelica. One can say that RML was our first approximation of the language subset to be used for bootstrapping. We then developed the MetaModelica 1.0 frontend to the RML kernel compiler and automatically translated the Modelica specification from RML to MetaModelica 1.0, to make it a better approximation of the language subset. Finally we extended the Modelica specification (i.e., the OpenModelica implementation), to add definitions of the needed metamodeling extensions, achieving the MetaModelica 2.0 language described in this report. Updating the Modelica specification, i.e., the OpenModelica compiler source code, to take full advantage of the MetaModelica 2.0 language still remains at the time of this writing.

The bootstrapping of the OpenModelica Compiler (OMC) has been a 5-year part-time effort, consisting of the following stages:

1. Design of an early MetaModelica language version (Fritzson, Pop, Aronsson 2005; Pop, Fritzson 2006) as an extended subset of Modelica, spring 2005.

2. Implementation of a MetaModelica Compiler (MMC) by adding a new compiler frontend to the old RML compiler (Pettersson 1999; Fritzson, Pop, Broman, Aronsson 2009), for the capability of compiling MetaModelica into RML intermediate form, spring-fall 2005.
3. Automatically translating the whole OpenModelica compiler, 60 000 lines, from RML to MetaModelica (Carlsson 2005).
4. In parallel, developing a new Eclipse plugin, MDT (Modelica Development Tooling), for Modelica and MetaModelica (Pop, Fritzson, Remar, Jagudin, Akhvlediani 2006; Pop 2008), including both browsing, debugging, semantic context-sensitive information, etc., 2005-2006.
5. Switching to using this MetaModelica 1.0 (an extended subset of Modelica), the MMC compiler, and the new MDT Eclipse plugin for the OpenModelica compiler development, 3-4 full-time developers. This version 1.0 of MetaModelica is described in (Fritzson 2007; Fritzson and Pop 2011) fall 2006.
6. Preliminary implementation of pattern-matching (Stavåker, Pop, Fritzson 2008) and exception handling (Pop, Stavåker, Fritzson 2008) in the OpenModelica compiler, to enable future bootstrapping. Spring-fall 2008.
7. Continuation of the work on better support for pattern-matching compilation, support for lists, tuples, records, etc. in OpenModelica, as part of metamodeling support in the OMC Java interface (Sjölund 2009) Spring-fall 2009.
8. Implementation of function arguments to functions (used in MetaModelica since 2005), also in OpenModelica (Brus 2011). This also became part of the Modelica 3.2 standard (Modelica Association 2010). Fall 2009, spring 2010.
9. Further adding, enhancing, and redesigning the MetaModelica language features based on usage experience, the Modelica 4 design effort, and inspiration from functional languages such as Standard ML (Milner, Tofte, Harper, and MacQueen 1997) and (Wikipedia-OCAML 2011) as well as languages such as Scala (Odersky, Spoon, and Venners 2008). Parts of the compiler will be refactored to use the enhanced features. This is currently ongoing work. The first enhanced version of MetaModelica is called MetaModelica 2.0 and is described in this report.

Chapter 2

Expression Evaluators and Interpreters in MetaModelica

We will introduce the topic of language specification in MetaModelica through a number of example languages.

The reader who would first prefer a general overview of some language properties of the MetaModelica subset for language specification may want to read Chapter 5 before continuing with these examples. On the other hand, the reader who has no previous experience with formal semantic specification and is more interested in "hands-on" use of MetaModelica for language implementation is recommended to continue directly with the current chapter and later take a quick glance at those chapters.

First we present a very small expression language called Exp1.

2.1 The Exp1 Expression Language

A very simple expression evaluator (interpreter) is our first example. This calculator evaluates constant expressions such as:

```
12 + 5 * 3
```

or

```
-5 * (10 – 4)
```

The evaluator accepts text of a constant expression, which is converted to a sequence of tokens by the lexical analyzer (e.g. generated by Lex or Flex) and further to an abstract syntax tree by the parser (e.g. generated by Yacc or Bison). Finally the expression is evaluated by the interpreter (generated by the MetaModelica compiler), which in the above case would return the value 27. This corresponds to the general structure of a typical interpreter as depicted in Figure 1-2.

2.1.1 Concrete Syntax

The concrete syntax of the small expression language is shown below expressed as BNF rules in Yacc style, and lexical syntax of the allowed tokens as regular expressions in Lex style. All token names are in upper-case and start with `T_` to be easily distinguishable from nonterminals which are in lower-case.

```
/* Yacc BNF Syntax of the expression language Exp1 */

expression       :  term
                 |  expression  weak_operator  term

term             :  u_element
                 |  term  strong_operator  u_element

u_element        :  element
```

```
                |  unary_operator  element

element         :  T_INTCONST
                |  T_LPAREN  expression  T_RPAREN

weak_operator   :  T_ADD   |  T_SUB
strong_operator :  T_MUL   |  T_DIV
unary_operator  :  T_SUB


/* Lex style lexical syntax of tokens in the expression language Exp1 */

digit       ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9")
digits      {digit}+
%%
{digits}    return T_INTCONST;
"+"         return T_ADD;
"-"         return T_SUB;
"*"         return T_MUL;
"/"         return T_DIV;
"("         return T_LPAREN;
")"         return T_RPAREN;
```

Lex also allows a more compact notation for a set of alternative characters which form a range of characters, as in the shorter but equivalent specification of digit below:

```
digit       [0-9]
```

2.1.2 Abstract Syntax of Exp1 with Union Types

The role of abstract syntax is to convey the structure of constructs of the specified language. It abstracts away (removes) some details present in the concrete syntax, and defines an unambiguous tree representation of the programming language constructs. There are usually several design choices for an abstract syntax of a given language. First we will show a simple version of the abstract syntax of the Exp1 language using the MetaModelica abstract syntax definition facilities.

2.1.3 The uniontype Construct

To be able to declare the type of abstract syntax trees we introduce the `uniontype` construct into Modelica:

- A union type specifies a union of one or more record types.
- Its record types and constructors are currently imported into the surrounding scope. This is planned to be changed in MetaModelica 3.0 to explicit import.
- Union types can be recursive – they can reference themselves.

A common usage is to specify the types of abstract syntax trees. In this particular case the following holds for the `Exp` union type:

- The `Exp` type is a union type of six record types
- Its record constructors are `INTConst`, `ADDop`, `SUBop`, `MULop`, `DIVop`, and `NEGop`.

The `Exp` union type is declared below. Its constructors are used to build nodes of the abstract syntax trees for the Exp language.

```
/* Abstract syntax of the language Exp1 as defined using MetaModelica */

uniontype Exp
  record  INTconst  Integer int;       end INTconst;
  record  ADDop  Exp exp1;  Exp exp2;  end ADDop;
  record  SUBop  Exp exp1;  Exp exp2;  end SUBop;
  record  MULop  Exp exp1;  Exp exp2;  end MULop;
```

```
  record  DIVop  Exp exp1;  Exp exp2;  end DIVop;
  record  NEGop  Exp exp;               end NEGop;
end Exp;
```

Using the `Exp` abstract syntax definition, the abstract syntax tree representation of the simple expression 12 + 5 * 13 will be as shown in Figure 2-1. The `Integer` data type is predefined in MetaModelica. Other predefined MetaModelica data types are `Real`, `Boolean`, and `String` as well as the parametric type constructors `array`, `MRArray`, `List`, `Tuple`, and `Option`.

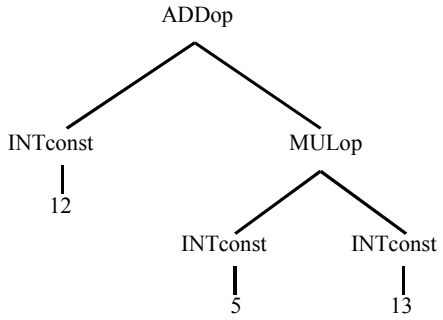

Figure 2-1. Abstract syntax tree of 12 + 5 * 13 in the language Exp1.

The `uniontype` declaration defines a union type `Exp` with constructors (in the figure: `ADDop`, `MULop`, `INTconst`) for each node type in the abstract syntax tree, as well as the types of the child nodes.

2.1.4 Semantics of Exp1

The semantics of the operations in the small expression language Exp1 follows below, expressed as an interpretive language specification in MetaModelica in a style reminiscent of Natural and/or Operational Semantics. Such specifications typically consist of a number of functions, each of which contains a match-expression with one or more cases. In this simple example there is only one function, here called `eval`, since we specify an expression evaluator.

2.1.4.1 Match-Expressions in MetaModelica

The following extension to Modelica is essential for specifying semantics of language constructs represented as abstract syntax trees:

- Match-expressions with pattern-matching cases, local declarations, and local equations.

A match-expression is closely related to pattern matching in functional languages, but is also related to switch statements in C or Java. It has two important advantages over traditional switch statements:

- A match-expression can appear in any of the three Modelica contexts: expressions, statements, or in equations.
- The selection in the case branches is based on pattern matching, which reduces to equality testing in simple cases, but is much more powerful in the general case.
- There are two variants of match-expressions using either the `match` or the `matchcontinue` keywords. The `match` keyword means that after a successful matching against a pattern in one of the case-branches no more patterns will be matched. The `matchcontinue` keyword means that even if there is a successful match followed by a failed computation in the same case-branch, the *matching will continue* with the subsequent case-branches.

A very simple example of a match-expression is the following code fragment, which returns a number corresponding to a given input string. The pattern matching is very simple – just compare the string

value of `s` with one of the constant pattern strings `"one"`, `"two"` or `"three"`, and if none of these matches return 0 since the wildcard pattern `_` (underscore) matches anything.

```
  String s;
  Integer x;
algorithm
  x := match s
    case "one"   then 1;
    case "two"   then 2;
    case "three" then 3;
    else         0;
  end match;
```

The else-branch is equivalent to and can be replaced with a `case _` using the `_` wildcard pattern that matches anything:

```
  String s;
  Integer x;
algorithm
  x := match s
    case "one"   then 1;
    case "two"   then 2;
    case "three" then 3;
    case _       then 0;
  end match;
```

Match-expressions have the following properties (see Section 7.2 for a more precise description):

- `match`. The value computed by the expression after the `match` keyword is matched against each of the patterns after the `case` keywords in order; if one matching fails the next is tried until there are no more case-branches in which case (if present) the else-branch is executed. *If a matching against a pattern succeeds but the rest of the computation in that case-branch fails, then the whole match-expression immediately fails*.
- `matchcontinue`. The value computed by the expression after the `matchcontinue` keyword is matched against each of the patterns after the `case` keywords in order; if one matching fails or if the matching succeeds but the computation in some part of the rest of the case fails, the next case (i.e., *matching continued*) is tried until there are no more case-branches in which case (if present) the else-branch is executed.
- Only algebraic equations are allowed as local equations in match-expressions, no differential equations.
- Only locally declared variables (local unknowns) declared by local declarations within the match-expression are solved for, and may appear as pattern variables.
- Equations are solved in the order they are declared.
- If an equation or an expression in a case-branch of a match-expression fails, all local variables become unbound, and matching continues with the next branch.

In the following we will primarily use match-expressions with `match` in the specifications.

2.1.4.2 Evaluation of the Exp1 Language

The first version of the specification of the calculator for the Exp1 language is using a rather verbose style, since we are presenting it in detail, including its explicit dependence on the pre-defined builtin semantic primitives such as integer arithmetic operations such as `intAdd`, `intSub`, `intMul`, etc. In the following we will show more concise versions of the specification, using the usual arithmetic operators which are just shorter syntax for the builtin arithmetic primitives.

```
function eval
  input  Exp     inExp;
  output Integer outInteger;
algorithm
  outInteger := match inExp
```

```
    local Integer v1,v2,v3;
          Exp     e1,e2;

    // evaluation of an integer node is the integer value itself
    case INTconst(v1) then v1;

    // Evaluation of an addition node ADDop is v3, if v3 is the result of
    // adding the evaluated results of its children e1 and e2
    // Subtraction, multiplication, division operators have similar specs.
    case ADDop(e1,e2)
      equation
        v1 = eval(e1); v2 = eval(e2); v3 = v1 + v2;
      then v3;

    case SUBop(e1,e2)
      equation
        v1 = eval(e1); v2 = eval(e2); v3 = v1 - v2;
      then v3;

    case MULop(e1,e2)
      equation
        v1 = eval(e1); v2 = eval(e2); v3 = intMul(v1,v2);
      then v3;

    case DIVop(e1,e2)
      equation
        v1 = eval(e1); v2 = eval(e2); v3 = intDiv(v1,v2);
      then v3;

    case NEGop(e1) equation
      v1 = eval(e1); v2 = intNeg(v1);  then v2;
   end match;

end eval;
```

In the `eval` function, which contains six cases, the first case has no constraint equations: it immediately returns a value.

```
    case INTconst(v1) then v1;   /* eval of an integer nodef */
```

This case states that the evaluation of an integer node containing an integer valued constant `int` will return the integer constant itself. The operational interpretation of the case is to match the argument to `eval` against the special case pattern `INTconst(v1)` for an integer constant expression tree. If there is a match, the pattern variable `v1` will be bound to the corresponding part of the tree. Then the local equations will be checked (there are actually no local equations in this case) to see if they are fulfilled. Finally, if the local equations are fulfilled, the integer constant value bound to `int` will be returned as the result.

We now turn to the second case of `eval`, which is specifying the evaluation of addition nodes labeled `ADDop`:

```
    case ADDop(e1,e2)
      equation
        v1 = eval(e1); v2 = eval(e2); v3 = v1 + v2;
      then v3;
```

For this case to apply, the pattern `ADDop(e1,e2)` must match the actual argument to `eval`, which in this case is an abstract syntax tree of the expression to be evaluated. If there is a match, the variables `e1` and `e2` will be bound the two child nodes of the `ADDop` node, respectively. Then the local equations of the case will be checked, in the order left to right. The first local equation states that the result of `eval(e1)` will be bound to `v1` if successful, the second states that the result of `eval(e2)` will be bound to `v2` if successful.

If the first two local equations are successfully solved, then the third local equation `v3 = v1+v2` will be checked. This local equation refers to a predefined operator or function `+` (same as `intAdd`) for addition of integer values. For a full set of predefined functions, including all common operations on

integers and real numbers, see Appendix B. This third local equation means that the result of adding integer values bound to `v1` and `v2` will be bound to `v3`. Finally, if all local equations are successful, `v3` will be returned as the result of the whole case.

The cases specifying the semantics of subtraction `-` (`SUBop`), multiplication `*` (`MULop`) and integer division `*` (`DIVop`) have exactly the same structure, apart from the fact that they map to different predefined operators such as `-(intSub)`, `*(intMul)`, and `/(intDiv)`.

The last case of the function `eval` specifies the semantics of a unary operator, unary integer negation, (example expression: -13):

```
    case NEGop(e1)
      equation
        v1 = eval(e1); v2 = intNeg(v1);
      then v2;
```

Here the expression tree `NEGop(e)` with constructor `NEGop` has only one subtree denoted by `e`. There are two local equations: the expression `e` should succeed in evaluating to some value `v1`, and the integer negation of `v1` will be bound to `v2`. Then the result of `NEGop(e)` will be the value `v2`.

It is possible to express the specification of the eval evaluator more concisely by using arithmetic operators such as +, -, *, etc., which is just different syntax for the builtin operations `intAdd`, `intSub`, `intMul`, and getting rid of the equations for the intermediate temporary variables `v1` and `v2`:

```
function eval
  input  Exp     inExp;
  output Integer outInteger;
algorithm
  outInteger := match inExp
    local Integer v1;  Exp e1,e2;
    case INTconst(v1) then v1;
    case ADDop(e1,e2) then eval(e1) + eval(e2);
    case SUBop(e1,e2) then eval(e1) - eval(e2);
    case MULop(e1,e2) then eval(e1) * eval(e2)
    case DIVop(e1,e2) then eval(e1) / eval(e2);
    case NEGop(e1)    then –eval(e1)
  end match;
end eval;
```

2.1.4.3 Using Named Pattern Matching for Exp1

So far we have used positional matching of values such as `inExp` to patterns such as `ADDop(e1,e2)`. The MetaModelica language also allows using *named pattern matching*, using the record field names of the corresponding record declaration to specify the pattern arguments. Thus, the pattern `ADDop(e1,e2)` would appear as `ADDop(exp1=e1,exp2=e2)` using named pattern matching. One advantage with named pattern matching is that only the parts of the pattern arguments that participate in the matching need to be specified. The wildcard arguments need not be specified.

The `Exp` uniontype with corresponding record declarations is shown again for clarity:

```
uniontype Exp
  record  INTconst  Integer int;       end INTconst;
  record  ADDop  Exp exp1;  Exp exp2;  end ADDop;
  record  SUBop  Exp exp1;  Exp exp2;  end SUBop;
  record  MULop  Exp exp1;  Exp exp2;  end MULop;
  record  DIVop  Exp exp1;  Exp exp2;  end DIVop;
  record  NEGop  Exp exp;              end NEGop;
end Exp;
```

Below we have changed all cases in the previous `eval` function example to use named pattern matching:

```
function eval
   input  Exp     inExp;
   output Integer outInteger;
algorithm
   outInteger := match inExp
     local Integer v1;  Exp e1,e2;
```

```
    case INTconst(int=v1)       then v1;
    case ADDop(exp1=e1,exp2=e2) then eval(e1) + eval(e2);
    case SUBop(exp1=e1,exp2=e2) then eval(e1) - eval(e2);
    case MULop(exp1=e1,exp2=e2) then eval(e1) * eval(e2)
    case DIVop(exp1=e1,exp2=e2) then eval(e1) / eval(e2);
    case NEGop(exp=e1)          then -eval(e1);
  end match;
end eval;
```

Furthermore, a compact version of pattern matching can be used that uses the record field names (e.g., `exp1`, `exp2`) via dot notation and avoids introducing extra pattern variables such as `e1`, `e2`.

The pattern `ADDop(__)`, see Section 7.3, with two underscores, matches all possible arguments to the corresponding constructor, here `ADDop`. The pattern `a as ADDop(__)`, see Section 7.2.4.2, means that the variable `a` becomes bound to what is matched with `ADDop(__)` with the record type of `ADDop`. The binding of the variable a to the corresponding record constructor is only accessible within the scope of the corresponding case branch. This is equivalent to the corresponding pattern `ADDop(exp1 = exp1, exp2 = exp2)`. See also Section 9.3.

```
function eval
  input  Exp     inExp;
  output Integer outInteger;
algorithm
  outInteger := match inExp
    local Exp a;
    case a as INTconst(__) then a.int;
    case a as ADDop(__)    then eval(a.exp1) + eval(a.exp2);
    case a as SUBop(__)    then eval(a.exp1) - eval(a.exp2);
    case a as MULop(__)    then eval(a.exp1) * eval(a.exp2);
    case a as DIVop(__)    then eval(a.exp1) / eval(a.exp2);
    case a as NEGop(__)    then -eval(a.exp);
  end match;
end eval;
```

Moreover, the input formal parameter `inExp` can be accessed directly using dot-notation since type-inference (Section 9.4) is used to deduce the specific record type of `inExp` in each case after matching. This is used in the following variant of the function `eval`. For example, within the `case INTconst(__)` the specific type of `inExp` in the scope of the case is inferred to be `Exp.INTconst` whereas in in the `case ADDop(__)` the specific type of `inExp` in the scope of the case is inferred to be `Exp.ADDop`.

```
function eval
  input  Exp     inExp;
  output Integer outInteger;
algorithm
  outInteger := match inExp
    case INTconst(__) then inExp.int;
    case ADDop(__)    then eval(inExp.exp1) + eval(inExp.exp2);
    case SUBop(__)    then eval(inExp.exp1) - eval(inExp.exp2);
    case MULop(__)    then eval(inExp.exp1) * eval(inExp.exp2);
    case DIVop(__)    then eval(inExp.exp1) / eval(inExp.exp2);
    case NEGop(__)    then -eval(inExp.exp);
  end match;
end eval;
```

2.2 Exp2 – Using Parameterized Abstract Syntax

An alternative, more parameterized style of abstract syntax is to collect similar operators in groups: all binary operators in one group, unary operators in one group, etc. The operator will then become a child of a `BINARY` node rather than being represented as the node type itself. This is actually more complicated than the previous abstract syntax for our simple language Exp1 but simplifies the semantic description of languages with many operators.

The Exp2 expression language is the same textual language as Exp1, but the specification uses the parameterized abstract syntax style which has consequences for the structure of both the abstract syntax and the semantic cases of the language specification.

We will continue to use the "simple" abstract representation in several language definitions, but switch to the parameterized abstract syntax for certain more complicated languages.

2.2.1 Parameterized Abstract Syntax of Exp1

Below is a parameterized abstract syntax for the previously introduced language Exp1, using the two nodes `BINARY` and `UNARY` for grouping. The Exp2 abstract syntax shown in the next section has the same structure, but with node constructors renamed to shorter names :

```
uniontype Exp
  record  INTconst Integer int;                      end INTconst;
  record  BINARY   Exp exp1; BinOp binOp; Exp exp2; end BINARY;
  record  UNARY    UnOp unOp; Exp exp;               end UNARY;
end Exp;

uniontype BinOp
  record  ADDop  end ADDop;
  record  SUBop  end SUBop;
  record  MULop  end MULop;
  record  DIVop  end DIVop;
end BinOp;

uniontype UnOp
  record  NEGop  end NEGop;
end BinOp;
```

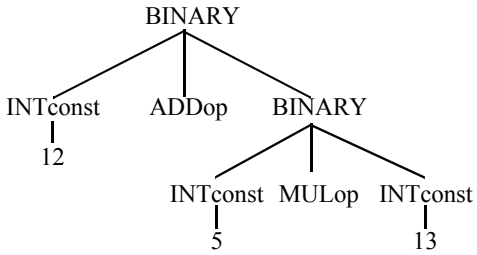

Figure 2-2. A parameterized abstract syntax tree of 12 + 5 * 13 in the language Exp1. Compare to the abstract syntax tree in Figure 2-1.

2.2.2 Parameterized Abstract Syntax of Exp2

Here follows the abstract syntax of the Exp2 language. The two node constructors `BINARY` and `UNARY` have been introduced to represent any binary or unary operator, respectively. Constructor names have been shortened to `INT`, `ADD`, `SUB`, `MUL`, `DIV` and `NEG`.

```
uniontype Exp
  record  INT     Integer int;                      end INT;
  record  BINARY  Exp exp1; BinOp binOp;  Exp exp2; end BINARY;
  record  UNARY   UnOp unOp; Exp exp;               end UNARY;
end Exp;

uniontype BinOp
  record  ADD  end ADD;
  record  SUB  end SUB;
  record  MUL  end MUL;
  record  DIV  end DIV;
end BinOp;

uniontype UnOp
  record  NEG  end NEG;
```

```
end BinOp;
```

2.2.3 Semantics of Exp2

As in the previous specification of Exp1, we specify the interpretive semantics of Exp2 via a series of cases expressed as case-branches in match-expressions comprising the bodies of the evaluation functions. However, first we need to introduce the notion of tuples in Modelica, since this is used in two of the evaluation functions.

2.2.3.1 Tuples in MetaModelica

Tuples are like records, but without field names. They can be used directly, without previous declaration of a corresponding tuple type.

The syntax of a tuple is a comma-separated list of values or variables, e.g. (..., ..., ...). The following is a tuple of a real value and a string value, using the tuple data constructor:

```
(3.14, "this is a string")
```

Tuples already exist in a limited way in previous versions of Modelica since functions with multiple results are called using a tuple for receiving results, e.g.:

```
(a, b, c) := foo(x, 2, 3, 5);
```

2.2.3.2 The Exp2 Evaluator

Below follows the semantic cases for the expression language Exp2, embedded in the functions `eval`, `applyBinop`, and `applyUnop`. As already mentioned, constructor names have been shortened compared to the specification of Exp1. Two cases have been introduced for the constructors `BINARY` and `UNARY`, which capture the common characteristics of all binary and unary operators, respectively. In addition to `eval`, two new functions `applyBinop` and `applyUnop` have been introduced, which describe the special properties of each binary and unary operator, respectively.

First we show the function header of the `eval` function, including the beginning of the match-expression:

```
function eval
  input Exp inExp;
  output Integer outInteger;
algorithm
  outInteger := match inExp
    local
      Integer ival,v1,v2,v3; Exp e1,e2,e;
      BinOp binop; UnOp unop;
```

Evaluation of an `INT` node gives the integer constant value itself:

```
    case INT(ival) then ival;
```

Evaluation of a binary operator node `BINARY` gives `v3`, if `v3` is the result of successfully applying the binary operator to `v1` and `v2`, which are the evaluated results of its children `e1` and `e2`:

```
    case BINARY(e1,binop,e2)
      equation
        v1 = eval(e1);
        v2 = eval(e2);
        v3 = applyBinop(binop, v1, v2);
      then v3;
```

Evaluation of a unary operator node `UNARY` gives `v2`, if its child `e` can be successfully evaluated to a value `v1`, and the unary operator can be successfully applied to value `v1`, giving the result value `v2`.

```
    case UNARY(unop,e)
      equation
        v1 = eval(e);
```

```
        v2 = applyUnop(unop, v1);
      then v2;

  end match;
end eval;
```

The Exp2 `eval` function can be made much more concise if we eliminate some intermediate variables and corresponding equations:

```
function eval
  input Exp inExp;
  output Integer outInteger;
algorithm
 outInteger := match inExp
    local
      Integer ival;  Exp e1,e2,e;
      BinOp binop;  UnOp unop;
    case INT(ival) then ival;
    case BINARY(e1,binop,e2) then applyBinop(binop, eval(e1), eval(e2));
    case UNARY(unop,e) then applyUnop(unop, eval(e));
  end match;
end eval;
```

Next to be presented is the function `applyBinop` which accepts a binary operator and two integer values.

```
function applyBinop
  input BinOp op;
  input Integer arg1;
  input Integer arg2;
  output Integer outInteger;
algorithm
  outInteger := match (op,arg1,arg2)
    local Integer v1,v2;
    case (ADD(),v1,v2) then v1 + v2;
    case (SUB(),v1,v2) then v1 - v2;
    case (MUL(),v1,v2) then v1 * v2;
    case (DIV(),v1,v2) then v1 / v2;
  end match;
end applyBinop;
```

If the passed binary operator successfully can be applied to the integer argument values an integer result will be returned. Note that we construct a tuple of three input values `(op,arg1,arg2)` in the match-expression which is matched against corresponding patterns in the case branches.

Finally we present the function `applyUnop` which accepts a unary operator and an integer value. If the operator successfully can be applied to this value an integer result will be returned.

```
function applyUnop
  input UnOp op;
  input Integer arg1;
  output Integer outInteger;
algorithm
  outInteger := match (op,arg1)
    local Integer v;
    case (NEG(), v) then –v;
  end match;
end applyUnop;
```

For the small language Exp2 the semantic description has become more complicated since we now need three functions, `eval`, `applyBinop` and `applyUnop`, instead of just `eval`. In the following, we will use the simple abstract syntax style for small specifications. The parameterized abstract syntax style will only be used for larger specifications where it actually helps in structuring and simplifying the specification.

2.3 Recursion and Failure in MetaModelica

Before continuing the series of language specifications expressed in MetaModelica, it is will be useful to say a few words about the MetaModelica language itself. A more in-depth description of the MetaModelica language constructs can be found in Chapter 5 to Chapter 11.

2.3.1 Short Introduction to Declarative Programming in MetaModelica

We have already stated that MetaModelica can be used as a declarative specification language for writing programming language specifications. Since Modelica is declarative, it can also be viewed as a functional programming language. A MetaModelica function containing match-expressions maps inputs to outputs, just as an ordinary function, but also has two additional properties:

- Functions containing match-expressions can succeed or fail.
- Local backtracking between cases can occur in match-expressions with `matchcontinue`. This means that if a case fails because one of its equations or function call `fail()` or has a run-time failure (e.g. division by zero, index out of bounds, etc.), the next case is tried.

The `fac` example below shows a function calculating factorials. This is an example of using MetaModelica to express a small declarative (i.e., functional) program:

```
function fac
  input  Integer inValue;
  output Integer outValue;
algorithm
 outValue := match inValue
    local  Integer n;
    case 0 then 1;
    case n then if n>0 then n*fac(n-1) else fail();
  end match;
end fac;
```

The first three lines specify the name (`fac`) and the type signature of the function. In this example an integer factorial function is computed, which means that both the input parameter and the result are of type `Integer`.

Next comes the two cases which make up the body of the match-expression in function. The first case in the above example can be interpreted as follows:

- If the function is called to compute the factorial of the value 0 (i.e. matching the "pattern" `fac(0)`), then the result is the value 1.

This corresponds to the base case of a recursive function calculating factorials.

The first case will be invoked if the argument matches the pattern `fac(0)` of the case. If this is not the case, the next case will be tried, if this case does not match, the next one will be tried, and so on. If no case matches the argument(s), the call to the function will fail.

The second case of the `fac` function handles the general case of a factorial function computation when the input value `n` is greater than zero, i.e., `n>0`. It can be interpreted as follows:

- If the factorial is computed on a value n, i.e., `fac(n)`, and `n>0`, then compute `n*fac(n-1)` which is returned as the result of the case.

2.3.2 Handling Failure

If the `fac` function is used to compute the factorial of a negative value an important property of MetaModelica is demonstrated, since the `fac` function will in this case fail.

A factorial call with a negative argument does not match the first case, since all negative values differs from zero. The second case matches, but fails, since the condition `n>0` is not fulfilled for negative values of n.

Thus the function will fail, meaning that it will not return an ordinary value to the calling function. After a `fail` has occurred in a match-expression (with matchcontinue) case or in some function called from that case, backtracking takes place, and the next case in the current match-expression (with matchcontinue) is tried instead. For match-expressions with the match keyword, there is no backtracking, i.e., after a fail has occurred in one of the cases the whole match-expression fails, except when the failing element is an argument to `failure`().

Functions with built-in failure handling can be useful, as in the following example:

```
function facFailsafe
  input Integer inValue;
  input Integer outValue;
algorithm
  outValue := matchcontinue inValue
    local
      Integer n,result; String str_result;

    case n
      equation
        str_result = intString(fac(n));
        print("Res: "); print(str_result);  print("\n");
      then 0;

    case n
      equation
        failure(result = fac(n));
        print("Cannot apply factorial relation to negative n."); print("\n");
      then 1;

  end matchcontinue;
end facFailsafe;
```

The function `facFailsafe` has two cases corresponding to the two cases of correct and incorrect arguments. Since the patterns are overlapping and we need to continue trying the next case, therefore using the matchcontinue variant of the match-expression. We use the **`failure`**(...) primitive to check for failure of the first equation in the second case.

The first case handles the case where the `fac` function computes the value and returns successfully. In this case the value is converted to a string and printed using the built-in MetaModelica `print` function.

The second case is tried if the first case fails, for example if the function `facFailsafe` is called with a negative argument, e.g. `fac(-1)`.

In the second case a new operator, **`failure`**`(...)`, is introduced in the expression `failure(result = fac(n))` which succeeds if the call `fac(n)` fails. Then an error message is printed by the second case.

It is important to note that `fail` is quite different from returning the logical value `false`. A function returning `false` would still succeed since it returns a value. The builtin operator **`not`** operates on the logical values `true` and `false`, and is quite different from the **`failure`** operator. See also Section 10.6.4.

2.4 The Assignments Language – Introducing Environments

The Assignments language extends our simple evaluator with variables. For example, the assignment:

```
a := 5 + 3 * 10
```

will store the value of the evaluated expression (here 35) into the variable `a`. The value of this variable can later be looked up and used for computing other expressions:

```
b := 100 + a
d := 10 * b
```

giving the values 135 and 1350 for `b` and `d`, respectively. Expressions may also contain embedded assignments as in the example below:

```
e := 50 + (d := a + 100)
```

2.4.1 Environments

To handle variables, we need a mechanism for associating values with identifiers. This mapping from identifiers to values is called an environment, and can be represented as a set of pairs (*identifier*,*value*). A function called `lookup` is introduced for looking up the associated value for a given identifier. An association of some value or other structure to an identifier is called a binding. An identifier is bound to a value within some environment.

There are several possible choices of data structures for representing environments. The simplest representation, often used in formal specifications, is to use a linked list of (*identifier*,*value*) pairs. This has the advantage of simplicity, but gives long lookup times due to linear search if there are many identifiers in the list. Other, more complicated, choices are binary trees or hash tables. Such representations are commonly used to provide fast lookup in product quality compilers or interpreters.

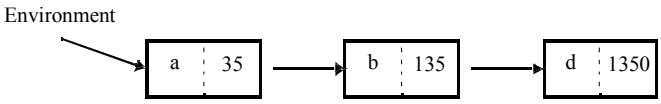

Figure 2-3. An environment represented as a linked list, containing name-value pairs for a, b and d.

Here we will regard the environment as an abstract data structure only accessed through access functions such as `lookup`, to avoid exposing specific low level implementation details. This gives us freedom to change the underlying implementation without changing the language specification. Unfortunately, many published formal language specifications have exposed such details and made themselves dependent on a linked list implementation. In the following we will initially use a linked list implementation of the environment abstract data type, which could be changed in the future when generating production quality translators.

In this simple Assignments language, an integer value is stored in the environment for each variable. Compilers need other kinds of values such as descriptors, containing various information for example location, type, length, etc., associated to each name. Compilers also use more complicated structures, called symbol tables, to store information associated with names. An environment can be regarded as a simplified abstract view of the symbol table.

2.4.2 Concrete Syntax of the Assignments Language

The concrete syntax of the Assignments language follows below. A couple of new cases have been added compared to the Exp language: one case for the assignment statement, two cases for the sequence of assignments, one rule for allowing assignments as subexpressions, and finally the program production has been extended to first take a sequence of assignments, then a separating semicolon, and lastly an ending expression.

```
/* Yacc BNF grammar of the expression language called Assignments */

program          : assignments T_SEMIC expression

assignments      :  assignment
                 |  assignments  assignment
```

```
assignment       :  ident  T_ASSIGN  expression

expression       :  term
                 |  expression  weak_operator  term

term             :  u_element
                 |  term  strong_operator  u_element

u_element        :  element
                 |  unary_operator  element

element          :  T_INTCONST
                 |  T_LPAREN  expression  T_RPAREN
                 |  T_LPAREN assignment T_RPAREN

weak_operator    :  T_ADD   |  T_SUB
strong_operator  :  T_MUL   |  T_DIV
unary_operator   :  T_SUB
```

The lexical specification for the Assignments language contains three more tokens, `":="`, `ident`, and `";"`, compared to the Exp1 language. It is a more complete lexical specification, making extensive use of regular expressions.

White space represents one or more blanks, tabs or new lines, and is ignored, i.e., no token is returned. A `letter` is a letter a-z or A-Z or underscore. An identifier (`ident`) is a letter followed by zero or more letters or digits. A `digit` is a character within the range 0-9. `digits` consists one or more of digit. An integer constant (`intcon`) is the same as `digits`. The function `lex_ident` returns the token `T_IDENT` and converts the scanned name to an atom representation stored in the global variable `yylval.voidp` which is used by the parser to obtain the identifier. The function `lex_icon` returns the token `T_INTCONST` and stores the integer constant converted into binary form in the same `yyval.voidp`.

```
/* Lex style lexical syntax of tokens in the language Assignments */

whitespace   [ \t\n]+
letter       [a-zA-Z_]
ident        {letter} ({letter} | {digit})*
digit        [0-9]
digits       {digit}+
%%
{whitespace} ;
{ident}            return lex_ident(); /* T_IDENT */
{digits}           return lex_icon();  /* T_INTCONST */
":="         return T_ASSIGN;
"+"          return T_ADD;
"-"          return T_SUB;
"*"          return T_MUL;
"/"          return T_DIV;
"("          return T_LPAREN;
")"          return T_RPAREN;
";"          return T_SEMIC;
```

2.4.3 Abstract Syntax of the Assignments Language

We introduce a few additional node types compared to the Exp1 language: the `ASSIGN` constructor representing assignment and the `IDENT` constructor for identifiers.

```
uniontype Exp
  record  INT     Integer integer;                    end INT;
  record  IDENT   Ident ident;                        end IDENT;
  record  BINARY  Exp exp1;  BinOp binOp;  Exp exp2; end BINARY;
  record  UNARY   UnOp unOp;  Exp exp;                end UNARY;
  record  ASSIGN  Ident ident;  Exp exp;              end ASSIGN;
```

```
end Exp;
```

Now we have also added a new abstract syntax type `Program` that represents an entire program as a list of assignments followed by an expression:

```
uniontype Program
  record  PROGRAM  ExpLst expLst;  Exp exp;  end PROGRAM;
end Program;

type  ExpLst     = List<Exp>;
```

The first list of expressions contains the initial list of assignments made before the ending expression will be evaluated.

The new type `Ident` is exactly the same as the builtin Modelica type `String`. The Modelica `type` declaration just introduces new names for existing types. The type `Value` is the same as `Integer` and represents integer values.

```
type Ident       =  String;
type Value       =  Integer;
```

The environment type `Env` is represented as a list of pairs (tuples) of (*identifier*,*value*) representing bindings of type `VarBnd` of identifiers to values. The MetaModelica syntax for tuples is: (*item1*, *item2*, ... *itemN*) of which a pair is a special case with two items. The MetaModelica `List` type constructor denotes a list type.

```
type VarBnd        = Tuple<Ident,Value>;
type Env           = List<VarBnd>;
```

Below follows all abstract syntax declarations needed for the specification of the Assignments language.

```
/* Complete abstract syntax for the Assignments language */

uniontype Exp
  record  INT     Integer integer;  end INT;
  record  IDENT   Ident ident;  end IDENT;
  record  BINARY  Exp exp1;  BinOp binOp;  Exp exp2;  end BINARY;
  record  UNARY   UnOp unOp;  Exp exp;  end UNARY;
  record  ASSIGN  Ident ident;  Exp exp;  end ASSIGN;
end Exp;

uniontype BinOp
  record  ADD  end ADD;
  record  SUB  end SUB;
  record  MUL  end MUL;
  record  DIV  end DIV;
end BinOp;

uniontype UnOp
  record  NEG  end NEG;
end BinOp;

uniontype Program
  record  PROGRAM  ExpLst expLst;  Exp exp;  end PROGRAM;
end Program;

type ExpLst        = List<Exp>;
type Ident         = String;

/* Values stored in environments */
type Value         = Integer;

/* Bindings and environments */
type VarBnd        = Tuple<Ident,Value>;
type Env           = List<VarBnd>;
```

2.4.4 Semantics of the Assignments Language

As previously mentioned, the Assignments language introduces the treatment of variables and the assignment statement to the former Exp2 language. Adding variables means that we need to remember their values between one expression and the next. This is handled by an environment (also known as evaluation context), which in our case is represented as list of variable-value pairs.

A semantic case will evaluate each descendent expression in one environment, modify the environment if necessary, and then pass the value of the expression and the new environment to the next evaluation.

2.4.4.1 Semantics of Lookup in Environments

To check whether an identifier is already present in an environment, and if so, return its value, we introduce the function `lookup`, see also Section 2.5.4.8. If there is no value associated with the identifier, `lookup` will fail.

```
function lookup
  input Env inEnv;
  input Ident inIdent;
  output Value outInteger;
algorithm
 outInteger := match (inEnv, inIdent)
    local Ident id2,id; Value value; Env rest;
    case ( (id2, value) :: rest, id)
      then
        if id == id2
        then value
        else lookup(rest,id);
  end match;
end lookup;
```

This version of `lookup` performs a linear search of an environment represented as a list of pairs (*identifier*,*value*).

The case works as follows: Either identifier `id` is found (`id == id2`) in the first pair of the list, and `value` is returned, or it is not found in the first pair of the list, and `lookup` will recursively search the `rest` of the list. If found, `value` is returned, otherwise the function will fail since there is no match.

In more detail, the pattern `(id2, value) :: rest` is matched against the environment argument `inEnv`. The `::` is the `cons` operator for adding a new element at the front of a list; and `rest` is a pattern variable the becomes bound to the rest of the list. If there is a match, `id2` will become bound to the identifier of that pair, and `value` will be bound to its associated value. If the condition `id == id2` is fulfilled, then `value` will be returned as the result of `lookup`, otherwise a recursive call to `lookup` is performed.

For example, the environment (`env`) depicted in Figure 2-3 shown is below:

```
{ (a, 35), (b, 135), (d, 1350) }
```

The list is the result of several `cons` operations:

```
(a, 35) :: (b, 135) :: (d, 1350) :: {}
```

An example lookup call:

```
lookup(env, a)
```

will match the pattern

```
lookup( (id2, value) :: rest, id)
```

This pattern belongs to the first case, and thereby binds `id2` to `a`, `value` to 35, `id` to `a`, and `rest` to `{(b,135),(d,1350)}`. Since the condition `id==id2` is fulfilled, the value 35 will be returned.

Below we also show a slightly more complicated variant of `lookup`, which does the same job, but is interesting from a semantic point of view. It has two cases. Since the patterns are overlapping and we need to continue trying the next case if the first case fails.

```
function lookup
  input Env inEnv;
  input Ident inIdent;
  output Value outValue;
algorithm
 outValue := matchcontinue (inEnv,inIdent)
    local  Ident id2,id;  Value value;  Env rest;

    // Identifier id is found in the first pair of
    // the list, and value is returned.
    case ( (id2, value) :: _, id)
      equation
        true = id == id2;
      then value;

    // Identifier id is not found in the first pair of the list, and lookup will
    // recursively search the rest of the list. If found, value is returned.
    case ( (id2, _) :: rest, id)
      equation
        false = id == id2; value = lookup(rest, id);
      then value;

  end matchcontinue;
end lookup;
```

The *first* case, also shown below, deals with the case when the identifier is present in the leftmost (most recent) pair in the environment.

```
    case ( (id2, value) :: rest ,id)
      equation
        true = id == id2;
      then value;
```

It will try to match the `(id2, value) :: rest` pattern against the environment argument. The `rest` is a pattern variable that will be bound to the rest of the environment represented as a list of (`id`, `value`) pairs. If there is a match, `id2` will become bound to the identifier of that pair, and `value` will be bound to its associated value. If `id == id2`, then `value` will be returned as the result of `lookup`, otherwise the identifier might be present in the rest of the list (i.e., not in the leftmost pair) and search continues in the rest of the list through the recursive call `lookup(rest,id)`.

For a call such as `lookup(env, b)`, `id2` will be bound to `a`, `rest` to `{ (b, 135), (d, 1350) }`, and `id` to `b`.

The first local equation of the second case below states that `id` is not in the leftmost pair (`(a,35)` in the above example call), whereas the second local equation retrieves the value from the rest of the environment if it succeeds.

```
    case ( (id2, _) :: rest, id )
      equation
        false = id==id2; value = lookup(rest, id);
      then value;
```

2.4.4.2 Updating and Extending Environments at Lookup

In the Assignments language we have the following two cases for the occurrence of an identifier (i.e., a variable) in an expression:

- If the variable is not yet in the environment, initialize it to zero and return its zero value and the new environment containing the added variable.
- If the variable is already in the environment, return its value together with the environment.

This is expressed by the function `lookupextend` below:

```
function lookupextend
  input Env inEnv;
  input Ident inIdent;
```

```
  output Env outEnv;
  output Value outValue;
algorithm
 (outEnv, outValue) := matchcontinue (inEnv,inIdent)
    local  Env env;  Ident id;  Value value;

    case (env,id)
      equation
        failure(value = lookup(env, id));
      then ( (id,0) :: env), 0);

    case (env,id)
      equation
        value = lookup(env, id);
      then (env, value);

  end matchcontinue;
end lookupextend;
```

For example, consider the following call on the above example environment `env`:

```
lookupextend(env,x)
```

This call will return the following environment together with the value 0:

```
{ (x, 0), (a, 35), (b, 135), (d, 1350) }
```

For the sake of completeness, we also show a version of `lookupextend` with two cases corresponding to the above two cases concerning the occurrence of an identifier. Both cases are using the same pattern `(env,id)`. Here we need to continue matching with the next cases if the current case fails – a kind of exception handling for fail exceptions. A match-expression would immediately return with a fail if the current case fails.

```
function lookupextend
  input Env inEnv;
  input Ident inIdent;
  output Env outEnv;
  output Value outValue;
algorithm
 (outEnv, outValue) := matchcontinue (inEnv,inIdent)
    local  Env env;  Ident id;  Value value;

    case (env, id)
      equation
        failure(value = lookup(env, id));
      then ( (id, 0) :: env, 0 );

    case (env, id)
      equation
        value = lookup(env, id);
      then (env, value);

  end matchcontinue;
end lookupextend;
```

For the evaluation of an assignment (node `ASSIGN`) we need to store the variable and its value in an updated environment, expressed by the following two cases:

- If the variable on the left hand side of the assignment is not yet in the environment, associate it with the value obtained from evaluating the expression on the right hand side, store this in the environment, and return the new value and the updated environment.
- If the variable on the left hand side is already in the environment, replace the current variable value with the value from the right hand side, and return the new value and the updated environment.

We actually cheat a bit in the function `update` below. Both `lookupextend` and `update` add a new pair `(id, value)` at the front of the environment represented as a list, even if the variable is already present. Since `lookup` will always search the environment association list from beginning to end, it will always return the most recent value, which gives the same semantics in terms of computational behavior but consumes more storage than a solution which would locate the existing pair and replace the value. The function `update` is as follows:

```
function update
  input Env inEnv;
  input Ident inIdent;
  input Value inValue3;
  output Env outEnv;
algorithm
 outEnv := matchcontinue (inEnv,inIdent,inValue3)
    local Env env;  Ident id;  Integer value;
    case (env,id,value) then (id,value) :: env;
  end matchcontinue;
end update;
```

For example, the following call is to update the variable `x` in the above example environment `env`:

```
update(env, x, 999)
```

This call will give the following environment list:

```
{ (x, 999), (a, 35), (b, 135), (d, 1350) }
```

One more call `update(env,x,988)` on the returned environment will give:

```
{ (x, 988), (x, 999), (a, 35), (b, 135), (d, 1350) }
```

Regard a call to lookup the variable `x` in the new environment (here called `env3`):

```
lookup(env3, x)
```

This call will return the most recent value of `x`, which is 988.

2.4.4.3 Evaluation Semantics

The `eval` function from the earlier Exp2 language has been extended with cases for assignment (`ASSIGN`) and variables (`IDENT`), as well as accepting an environment as an extra argument and returning an (updated) environment as a result. In the case to evaluate an `IDENT` node, `lookupextend` returns a possibly updated environment `env2` and the value associated with the identifier `id` in the current environment `env`. If there is no such value, identifier `id` will be bound to zero and the current environment will be updated to become `env2`.

```
function eval
  input Env inEnv;
  input Exp inExp;
  output Env outEnv;
  output Integer outInteger;
algorithm
  (outEnv, outInteger) := matchcontinue (inEnv,inExp)
    local
      Env env,env1,env2,env3;
      Integer ival, value, v1,v2,v3;
      Ident id;
      Exp exp,e1,e2,e;
      BinOp binop;  UnOp unop;

    // eval of an integer constant node INT in an environment is the integer
    // value together with the unchanged environment.
    case (env,INT(ival))   then (env,ival);

    // eval of an identifier node IDENT will lookup the identifier and return a
    // value if present; otherwise insert a binding to zero, and return zero.
    case (env,IDENT(id))
```

```
      equation
        (env2,value) = lookupextend(env, id);
      then (env2,value);

    // eval of an assignment node returns the updated environment and
    //  the assigned value.
    case (env,ASSIGN(id,exp))
      equation
        (env2,value) = eval(env, exp);
        env3 = update(env2, id, value);
      then (env3,value);
```

The cases below specify the evaluation of the binary (`ADD`, `SUB`, `MUL`, `DIV`) and unary (`NEG`) operators. The first case specifies that the evaluation of an binary node `BINARY(e1,binop,e2)` in an environment env1 is a possibly changed environment `env3` and a value `v3`, provided that function `eval` succeeds in evaluating `e1` to the value `v2` and possibly a new environment `env2`, and `e2` successfully evaluates `e2` to the value `v2` and possibly a new environment `env3`. Finally, the `applyBinop` function is used to apply the operator to the two evaluated values. The reason for returning new environments is that expressions may contain embedded assignments, for example: `e := 35 + (d := a + 100)`. The case for unary operators is similar.

```
    // eval of a binary node BINARY(e1,binop,e2), etc. in an environment env
    case (env1,BINARY(e1,(binop,e2)))
      equation
        (env2,v1) = eval(env1, e1);
        (env3,v2) = eval(env2, e2);
        v3 = applyBinop(binop, v1, v2);
      then (env3,v3);

    // eval of a unary node UNARY(unop,e), etc. in an environment env
    case (env1,UNARY(unop,e))
      equation
        (env2,v1) = eval(env1, e);
        v2 = applyUnop(unop, v1);
      then (env2,v2);

  end matchcontinue;
end eval;
```

The functions `applyBinop` and `applyUnop` are not shown here since they are unchanged from the Exp2 specification.

In Section 2.6 the Assignments language will be extended into a language called AssignTwoType that can handle expressions containing constants and variables of two types: `Real` and `Integer`, which has interesting consequences for the semantics of the evaluation cases and storing values in the environment.

2.5 PAM – Introducing Control Structures and I/O

PAM is a Pascal-like language that is too small to be useful for serious programming, but big enough to illustrate several important features of programming languages such as control structures, including loops (but excluding goto), and simple input/output. However, it does not include procedures/functions and multiple types. Only integer variables and values are dealt with during computation, although `Boolean` values can occur temporarily in comparisons within if- or while-statements.

2.5.1 Examples of PAM Programs

A PAM program consists of a series of statements, as in the example below where the factorial of a number `N` is computed. First the number `N` is read from the input stream. Then the special case of

factorial of zero is dealt with, giving the value 1. Note that factorial of a negative number is not handled by this program, not even by an error message since there are no strings in this language.

The factorial for `N>0` is computed by the else-part of the if-statement, which contains a definite loop:

`to` *expression* **`do`** *series-of-statement* **`end`**

This loop computes *series-of-statemen*t a definite number of times given by first evaluating *expression*. In the example below, **`to`** `N` **`do ... end`** will compute the factorial by iterating `N` times. Alternatively, we could have expressed this as an indefinite loop, i.e., a while statement:

`while` *comparison* **`do`** *series-of-statement* **`end`**

The while statement will evaluate *series-of-statement* as long as *comparison* is true.

```
/* Computing factorial of the number N, and store in variable Fak */
/* N is read from the input stream; Fak is written to the output */
/* Fak is  1 * 2 * ....  (N-1) * N    */
read N;
if N=0 then
  Fak := 1;
else
  if N>0 then
    Fak := 1;
    I := 0;
    to N do
      I := I+1;
      Fak := Fak*I;
    end
  endif
endif
write Fak;
```

Variables are not declared in this language, they are created when they are assigned values. The usual arithmetic operators "+", "-" with weak precedence and "*", "/" with stronger precedence, are included. Comparisons are expressed by the relational operators "<", "<=", "=", ">=", ">". One small change has been done to PAM as compared to Pagan's book: the reserved word `FI` has been replaced by the more readable `endif`.

2.5.2 Concrete Syntax of PAM

The concrete syntax of the PAM language is given as a BNF grammar below. A program is a *series_of_statement*. A *statement* is an *input_statement* (**`read`** `id1,id2,...`); an *output_statement* (**`write`** `id1,id2...`); an *assignment_statement* (`id :=` *expression*); an if-then *conditional_statement* (**`if`** *expression* **`then`** *series-of-statement* **`endif`**), an if-then-else *conditional_statement* (**`if`** *expression* **`then`** *series-of-statement* **`else`** *series-of-statement* **`endif`**), a *definite_loop* for a fixed number of iterations (**`to`** *expression* **`do`** *series-of-statement* **`end`**), or a *while_loop* for an indefinite number of iterations (**`while`** *comparison* **`do`** *series-of-statement* **`end`**). The usual arithmetic expressions are included, as well as comparisons using relational operators.

```
/* Yacc BNF grammar of the PAM language */

program               :  series

series                :  statement
                      |  statements series

statement             :  input_statement T_SEMIC
                      |  output_statement T_SEMIC
                      |  assignment_statement T_SEMIC
                      |  conditional_statement
                      |  definite_loop
                      |  while_loop
```

```
input_statement       :  T_READ  variable_list

output_statement      :  T_WRITE  variable_list

variable_list         :  variable
                      |  variable variable_list

assignment_statement  :  variable  T_ASSIGN  expression

conditional_statement :  T_IF comparison T_THEN series T_ENDIF
                      |  T_IF comparison T_THEN series
                                       T_ELSE series T_ENDIF

definite_loop         |  T_TO expression T_DO series T_END

while_loop            |  T_WHILE comparison T_DO series T_END


expression       :  term
                 |  expression  weak_operator  term

term             :  element
                 |  term  strong_operator  element

element          :  constant
                 |  variable
                 |  T_LPAREN  expression  T_RPAREN

comparison       :  expression  relation  expression

variable         :  T_IDENT
constant         :  T_INTCONST
relation         :  T_EQ | T_LE | T_LT  T_GT | T_GE | T_NE
weak_operator    :  T_ADD   |  T_SUB
strong_operator  :  T_MUL   |  T_DIV
```

The lexical syntax of the PAM language has two extensions compared to the previously presented Assignments language: tokens for relational operators “<”, “<=”, “=”, “<>”, “>=”, “>” and tokens for reserved words: `if`, `then`, `else`, `endif`, `while`, `do`, `end`, `to`, `read`, `write`. The function `lex_ident` checks if a possible identifier is a reserved word, and in that case returns one of the tokens `T_IF`, `T_THEN`, `T_ELSE`, `T_ENDIF`, `T_ELSE`, `T_WHILE`, `T_DO`, `T_END`, `T_TO`, `T_READ` or `T_WRITE`.

```
/* Lex style lexical syntax of tokens in the PAM language */

whitespace   [ \t\n]+
letter       [a-zA-Z]
ident        {letter} ({letter} | {digit})*
digit        [0-9]
digits       {digit}+
icon         {digits}
%%
{whitespace} ;
{ident}      return lex_ident(); /* T_IDENT or reserved word tokens */
/* Reserved words: if,then,else,endif,while,do,end,to,read,write */

{digits}     return lex_icon();  /* T_INTCONST */
":="         return T_ASSIGN;
"+"          return T_ADD;
"-"          return T_SUB;
"*"          return T_MUL;
"/"          return T_DIV;
"("          return T_LPAREN;
")"          return T_RPAREN;
"<"          return T_LT;
"<="         return T_LE;
"="          return T_EQ;
```

```
"<>"          return T_NE;
">="          return T_GE;
">"           return T_GT;
```

2.5.3 Abstract Syntax of PAM

Since PAM is slightly more complicated than previous languages we choose the parameterized style of abstract syntax, first introduced in Section 2.2 and Section 2.2. This style is better at grouping related semantic constructs and thus making the semantic specification more concise and better structured.

In comparison to the Assignments language, we have introduced relational operators (`RelOp`) and the `RELATION` constructor which belongs to the set of expression nodes (`Exp`). There is also a union type `Stmt` for different kinds of statements. Note that statements are different from expressions in that they do not return a value but update the value environment and/or modify the input or output stream. However, in this simplified semantics the streams are implicit and not part of the semantic model to be presented. The constructor `SEQ` allows the representation of statement sequences, whereas `SKIP` represents the empty statement.

```
/* Parameterized abstract syntax for the PAM language */

type Ident = String;

uniontype BinOp
  record ADD  end ADD;
  record SUB  end SUB;
  record MUL  end MUL;
  record DIV  end DIV;
end BinOp;

uniontype RelOp
  record EQ  end EQ;
  record GT  end GT;
  record LT  end LT;
  record LE  end LE;
  record GE  end GE;
  record NE  end NE;
end RelOp;

uniontype Exp
  record INT      Integer int;                      end INT;
  record IDENT    Ident ident;                      end IDENT;
  record BINARY   Exp exp1; BinOp binOp; Exp exp2; end BINARY;
  record RELATION Exp exp1; RelOp relOp; Exp exp2; end RELATION;
end Exp;

type IdentList = List<Ident>;

uniontype Stmt
  record ASSIGN Ident ident; Exp exp; end ASSIGN;        // Id := Exp
  record IF     Exp exp; Stmt stmt1; Stmt stmt2; end IF; // if Exp then Stmt..
  record WHILE  Exp exp; Stmt stmt;  end WHILE;          // while Exp do Stmt
  record TODO   Exp exp; Stmt stmt;  end TODO;           // to Exp do Stmt...
  record READ   IdentList identList;  end READ;         // read id1,id2,...
  record WRITE  IdentList identList;  end WRITE;        // write id1,id2,..
  record SEQ    Stmt stmt1;  Stmt stmt2;  end SEQ;      // Stmt1; Stmt2
  record SKIP  end SKIP;                                // ; empty stmt
end Stmt;
```

The type specifications below are not part of the abstract syntax of the language constructs, but needed to model the static and dynamic semantics of PAM. As for the Assignments language, the environment (`Env`) is a mapping from identifiers to values, used to store and retrieve variable values. Here it is represented as a list of pairs of variable bindings (`VarBnd`).

```
/* Types needed for modeling static and dynamic semantics */
```

```
/* Variable binding and environment/state type */
type VarBnd = Tuple<Ident,Value>;
type Env    = List<VarBnd>;
type Stream = List<Integer>;

type State  = Tuple<Env,Stream,Stream> "Environment,input stream,output stream";

uniontype Value  "Value type needed for evaluated results"
  record INTval  Integer intval;  end INTval;
  record BOOLval Boolean boolval; end BOOLval;
end Value;
```

We also introduce a union type `Value` for values obtained during expression evaluation. Even though only `Integer` values tagged by the constructor `INTval` are stored in the environment, `Boolean` values, represented by `BOOLval(Boolean)`, occur when evaluating comparison functions.

Since PAM contains input and output statements, we need to model the overall state including both variable bindings and input and output files. This could have been done (as in Pascal [ref **]) by introducing two predefined variables in the environment denoting the standard input stream and output stream, respectively. Since standard input/output streams are not part of the PAM language definition we choose another solution. The concept of state is introduced, of type `State`, which is represented as a triple of environment, input stream and output stream `(Env,Stream,Stream)`. The term configuration is sometimes used for this kind of state.

2.5.4 Semantics of PAM

The semantics of PAM is specified by several functions that contain groups of cases for similar constructs. Expression evaluation together with binary and relational operators are described first, since this is very close to previously presented expression languages. Then we present statement evaluation including simple control structures and input/output. Finally some utility functions for lookup of identifiers in environments, repeated evaluation, and I/O are defined.

2.5.4.1 Expression Evaluation

The `eval` function defines the semantics of expression evaluation. The first case specifies evaluation of integer constant leaf nodes (`INT(v)`) which evaluate independently of the environment (because of the underscore wildcard _) into the same constant value `v` tagged by the constructor `INTval`.

We choose to introduce a special data type `Value` with constructors `INTval` and `BOOLval` for values generated during the evaluation. Alternatively, we could have used the abstract syntax leaf node `INT`, and introduced another node called `BOOL`. However, we chose the `Value` alternative, in order not to mix up the type of values produced during evaluation with the node types of the abstract syntax. An additional benefit of giving the specification a more clear type structure is that the MetaModelica compiler will have better chances of detecting type errors in the specification.

```
function eval "Evaluation of expressions in the current environment"
  input Env inEnv;
  input Exp inExp;
  output Value outValue;
algorithm
  outValue := matchcontinue (inEnv,inExp)
    local
      Integer v,v1,v2,v3;
      Env env;
      Ident id;
      Exp e1,e2;
      BinOp binop;
      RelOp relop;
      Boolean bv3;

    case (_,INT(v)) then INTval(v);                        // Integer constant v
```

The next two cases define the evaluation of identifier leaf nodes (`IDENT(id)`). The first case describes successful lookup of a variable value in the environment, returning a tagged integer value (`INTval(v)`). The second case describes what happens if a variable is undefined. An error message is given and the evaluation will fail.

```
    case (env,IDENT(id)) then lookup(env, id);            // Identifier id

    case (env,IDENT(id))
      equation                          // If id not declared, give an error
        failure(v = lookup(env, id));  // message and fail by calling error()
      then error("Undefined identifier", id);
```

The last two cases specify evaluation of binary arithmetic operators and boolean relational operators, respectively. These cases first take care of argument evaluation, which thus need not be repeated for each case in the invoked functions `applyBinop` and `applyRelop` which compute the values to be returned. Here we see the advantages of parameterized abstract syntax, which allows grouping of constructs with similar structure. The last case returns values tagged `BOOLval`, which cannot be stored in the environment, and are used only for comparisons in while- and if-statements.

```
    case (env,BINARY(e1,binop,e2))                        // expr1 binop expr2
      equation
        INTval(v1) = eval(env, e1);
        INTval(v2) = eval(env, e2);
        v3 = applyBinop(binop, v1, v2);
      then INTval(v3);

    case (env,RELATION(e1,relop,e2))                      // expr1 relop expr2
      equation
        INTval(v1) = eval(env, e1);
        INTval(v2) = eval(env, e2);
        vv3 = applyRelop(relop, v1, v2);
      then BOOLval(bv3);

  end matchcontinue;
end eval;
```

2.5.4.2 Arithmetic and Relational Operators

The functions `applyBinop` and `applyRelop` define the semantics of applying binary arithmetic operators and binary boolean operators to integer arguments, respectively. Since argument evaluation has already been taken care of by the `eval` function, only one local equation is needed in each case to invoke the appropriate predefined MetaModelica operation.

```
function applyBinop
  "Apply a binary arithmetic operator to constant integer arguments"
  input BinOp op;
  input Integer arg1;
  input Integer arg2;
  output Integer outInteger;
algorithm
  outInteger := matchcontinue (op,arg1,arg2)
    local  Integer x,y;
    case (ADD(),x,y) then x + y;
    case (SUB(),x,y) then x – y;
    case (MUL(),x,y) then x * y;
    case (DIV(),x,y) then x / y;
  end matchcontinue;
end applyBinop;

function applyRelop
  "Apply a relation operator, returning a boolean value"
  input RelOp op;
  input Integer arg1;
  input Integer arg2;
  output Boolean outBoolean;
```

```
algorithm
  outBoolean := matchcontinue (op,arg1,arg2)
    local  Integer x,y;
    case (LT(),x,y) then (x < y);
    case (LE(),x,y) then (x <= y);
    case (EQ(),x,y) then (x == y);
    case (NE(),x,y) then (x <> y);
    case (GE(),x,y) then (x >= y);
    case (GT(),x,y) then (x > y);
  end matchcontinue;
end applyRelop;
```

2.5.4.3 Statement Evaluation

The `evalStmt` function defines the semantics of statements in the PAM language. In contrast to expressions, statements return no values. Instead they modify the current state which contains variable values, the input stream and the output stream. The type `State` is defined as follows:

```
type State  =  Tuple<Env,Stream,Stream>;
```

Statements change the current state, returning a new updated state. This is expressed by the type signature of `evalStmt` which is `(State, Stmt) => State`. Below we describe the function `evalStmt` by explaining the semantics of each statement type separately.

First we show the function header and the beginning of the match-expression

```
function evalStmt "Statement evaluation: map the current state into a new state"
  input State inState;
  input Stmt inStmt;
  output State outState;
algorithm
  outState := matchcontinue (inState,inStmt)
    local
      Value v1;
      Env env,env2;
      State state,state1,state2,state3;
      Stream istream,istream2,ostream,ostream2;
      Ident id;  Exp e1,comp;
      Stmt s1,s2,stmt1,stmt2;
      Integer n1,v2;
```

The semantics of an assignment statement `id := e1` is to first evaluate the expression `e1` in the current environment `env`, and then update `env` by associating identifier `id` with the value `v1`, giving a new environment `env2`. The returned state contains the updated environment `env2` together with unchanged input stream (`is`) and output stream (`os`).

```
    case (env,ASSIGN(id,e1))                              /* Assignment */
      equation
        v1 = eval(env, e1);
        env2 = update(env, id, v1);
      then env2;
```

The conditional statement occurs in two forms: a long form: `if` *comparison* `then` *stmt1* `else` *stmt2* or a short form `if` *comparison* `then` *stmt1*. Both forms are represented by the abstract syntax node (`IF(comp,s1,s2)`), where the short form has an empty statement (a `SKIP` node) in the else-part. Both `stmt1` and `stmt2` can be a sequence of statements, represented by the `SEQ` abstract syntax node.

The pattern `state1 as (env,_,_)` means that the state argument that matches `(env,_,_)` will also be bound to `state1`. The environment component of the state will be bound to `env`, whereas the input and output components always match because of the wildcards `(_,_)`.

The first case is where the comparison evaluates to true. Thus the then-part (statement `s1`) will be evaluated, giving a new state `state2`, which is the result of the if-statement. The second case covers the case where the comparison evaluates to false, causing the else-part (statement `s2`) to be evaluated, giving a new state `state2`, which then becomes the result of the if-statement.

```
    case (state1 as (env,_,_), IF(comp,s1,s2))             /* if true ... */
```

```
    equation
      BOOLval(true) = eval(env, comp);
      state2 = evalStmt(state1, s1);
    then state2;

  case (state1 as (env,_,_), IF(comp,s1,s2))            /* if false ... */
    equation
      BOOLval(false) = eval(env, comp);
      state2 = evalStmt(state1, s2);
    then state2;
```

These two cases can be compacted into one case, using a conditional expression:

```
  case (state as (env,_,_), IF(comp,s1,s2))             /* if ... */
    then
      if BOOLval(true) == eval(env, comp)
      then evalStmt(state, s1)
      else if BOOLval(false) == eval(env, comp)
           then evalStmt(state, s2)
           else fail();
```

The next case defines the semantics of the iterative while-statement. It is fundamentally different from all cases we have previously encountered in that the while construct recursively refers to itself in the local equation of the case. The meaning of while is the following: first evaluate the comparison `comp` in the current state. If true, then evaluate the statement (sequence) `s1`, followed by recursive evaluation of the while-loop. On the other hand, if the comparison evaluates to `false`, no further action takes place.

There are at least two ways to specify the semantics of while. The first version, shown in the case immediately below, uses the availability of if-statements and empty statements (`SKIP`) in the language. The if-statement will first evaluate the comparison `comp`. If the result is true, the then-branch will be chosen, which consists of a sequence of two statements. The while body (`s1`) will first be evaluated, followed by recursive evaluation of the while-loop once more. On the other hand, if the comparison evaluates to false, the else-branch consisting of the empty statement (`SKIP`) will be chosen, and no further action takes place.

Since the recursive invocation of while is tail-recursive (this occurs as the last action, at the end of the then-branch), the MetaModelica compiler can implement this case efficiently, without consuming stack space, similar to a conventional implementation that uses a backward jump. Note that this is only possible if there are no other candidate cases in the function.

```
  case (state,WHILE(comp,s1))                          // while ...
    equation
      state2 = evalStmt(state, IF(comp,SEQ(s1,WHILE(comp,s1)),SKIP()));
    then state2;
```

The semantics of the while-statement can alternatively be modeled by the two cases below. The first case, when the comparison evaluates to `false`, returns the current state unchanged. The second case, in which the comparison evaluates to `true`, subsequently evaluates the while-body (`s1`) once, giving a new state `state2`, after which the while-statement is recursively evaluated, giving the state `state3` to be returned.

```
  case (state as (env,_,_), WHILE(comp,s1))          // while false ...
    equation
      BOOLval(false) = eval(env,comp);
    then state;

  case (state as (env,_,_), WHILE(comp,s1))          // while true ...
    equation
      BOOLval(true) = eval(env,comp);
      state2 = evalStmt(state,s1);
      state3 = evalStmt(state2,WHILE(comp,s1));
    then state3;
```

Both versions of the while semantics are OK. Since the previous version is slightly more compact, using only one case, we choose that one in our final specification of PAM.

The definite iterative statement: `to` *expression* `do` *statement* `end` first evaluates expression `e1` to obtain some number `n1`, and provided that `n1` is positive, repeatedly evaluates statement `s1` the definite number of times given by `n1`. The repeated evaluation is performed by the function `repeatEval`.

```
case (state as (env,_,_), TODO(e1,s1))                    // to e1 do s1 ...
  equation
    INTval(n1) = eval(env, e1);
    state2 = repeatEval(state, n1, s1);
  then state2;
```

Read and write statements modify the input and output stream components of the state, respectively. The input stream and output streams can be thought of as infinite sequences of items (for PAM: sequences of integer constants), which are handled by the operating system. First we describe the `read` statement.

The read statement: `read` *id1*,*id2*,...*idN* reads *N* values into variables *id1*, *id2*,... *idN*, picking them from the beginning of the input stream which is updated as a result.

The first case covers the case of reading into an empty list of variables, which has no effect and returns the current state unchanged. The second case models actual reading of values from the input stream. First, one item is extracted from the input stream by calling `inputItem`, which returns a modified input stream and a value. The `inputItem` function should be regarded as part of an abstract interface that hides the implementation of `Stream`.

```
case (state,READ({})) then state;                         // read ()

case (state as (env,istream,ostream, READ(id :: rest))    // read id1,..
  equation
    (istream2,v2) = inputItem(istream);
    env2 = update(env, id, INTval(v2));
    state2 = evalStmt((env2,istream2,ostream), READ(rest));
  then state2;
```

Analogously, the write statement: `write` *id1*,*id2*,...*idN* writes *N* values from variables *id1*, *id2*,... *idN*, adding them to the end of the current output stream which is modified accordingly. Writing an empty list of identifiers has no effect.

```
case (state, WRITE({})) then state;                         // write ()

case (state as (env,istream,ostream), WRITE(id :: rest))    // write id1,..
  equation
    INTval(v2) = lookup(env, id);
    ostream2 = outputItem(ostream,v2);
    state2 = evalStmt((env,istream,ostream2), WRITE(rest));
  then state2;
```

The semantics of a sequence `stmt1; stmt2` of two statements is simple. First evaluate `stmt1`, giving an updated state `state2`. Then evaluate `stmt2` in `state2`, giving `state3` which is the resulting state.

```
case (state,SEQ(stmt1,stmt2))                               // stmt1 ; stmt2
  equation
    state2 = evalStmt(state, stmt1);
    state3 = evalStmt(state2, stmt2);
  then state3;
```

The semantics of the empty statement, represented as `SKIP`, is even simpler. Nothing happens, and the current state is returned unchanged.

```
case (state,SKIP()) then state;                            // ; empty statement

end matchcontinue;

end evalStmt;
```

2.5.4.4 Auxiliary Functions

The next few subsections define auxiliary functions, `repeatEval`, `error`, `inputItem`, `outputItem`, `lookup`, and `update`, needed by the rest of the PAM specification.

2.5.4.5 Repeated Statement Evaluation

The function `repeatEval(state,n,stmt)` simply evaluates the statement `stmt` `n` times, starting with state, which is updated into a new state for each iteration. The then-part specifies that nothing happens if `n` <= 0. The else-part evaluates `stmt` in `state` and recursively calls `repeatEval` for the remaining `n-1` iterations, giving state which is returned.

```
function repeatEval  "repeatedly evaluate stmt n times"
  input State state;
  input Integer n;
  input Stmt stmt;
  output State outState;
algorithm
  outState :=
    if n <= 0
    then state // n <= 0
    else repeatEval(evalStmt(state, stmt), n - 1, stmt); // eval n times
end repeatEval;
```

2.5.4.6 Error Handling

The `error` function can be invoked when there is some semantic error, for example when an undefined identifier is encountered. It simply prints one or two error messages, returns the empty value, and fails, which will stop evaluation (for an interpreter) or stop semantic analysis (for a translator).

```
function error  "Print error messages str1 and str2, and fail"
  input Ident str1;
  input Ident str2;
algorithm
  print("Error – ");
  print(str1);  print(" ");
  print(str2);  print("\n");
  fail();
end error;
```

2.5.4.7 Stream I/O Primitives

The `inputItem` function retrieves an item (here an integer constant) from the input stream, which can be thought of as an infinite list implemented by the operating system. The item is effectively removed from the beginning of the stream, giving a new (updated) stream consisting of the rest of the list. Since `Stream` in reality is implemented by the operating system, the streams passed to and returned from this implementation of `inputItem` and `outputItem` are not updated, they are just dummy streams which give the functions the correct type signatures.

```
function inputItem  "Read an integer item from the input stream"
  input Stream istream;
  output Stream istream2;
  output Integer i;
algorithm
  print("input: ");
  i := Input.read();
  print("\n");
  istream2 := istream;
end inputItem;
```

The `outputItem` function outputs an item by attaching the item to the front of the output stream (effectively a possibly infinite list of items), giving an updated output stream `ostream2`.

```
function outputItem  "Write an integer item on the output stream"
  input Stream ostream;
  input Integer i;
  output Stream ostream2;
 protected
```

```
  String s;
algorithm
  s := intString(i);
  print(s);
  ostream2 := ostream;
end outputItem;
```

2.5.4.8 Environment Lookup and Update

The function `lookup(env,id)` returns the value associated with identifier `id` in the environment `env`. If there is no binding for `id` in the environment, `lookup` will fail. Here the environment is implemented (as usual) as a linked list of (*identifier*,*value*) pairs.

The first case covers the case where `id` is found in the first pair of the list. The pattern `(id2,value)` is concatenated (`::`) to the rest of the list (the pattern wildcard: _), whereas the second case covers the case where `id` is not in the first pair, and therefore recursively searches the rest of the list.

```
function lookup
 "lookup returns the value associated with an identifier.
  If no association is present, lookup will fail."
  Input  Env   inEnv;
  input  Ident inIdent;
  output Value outValue;
algorithm
  outValue := matchcontinue (inEnv, inIdent)
    local  Ident id2,id;  Value value;  State rest;
    case ((id2, value) :: rest, id) then
      if id==id2
      then value              // id first in list
      else lookup(rest,id);   // id in rest of list
  end matchcontinue;
end lookup;
```

The function `update(env, id, value)` inserts a new binding between `id` and `value` into the environment. Here the new `(id, value)` pair is simply put at the beginning of the environment. If an existing binding of `id` was already in the environment, it will never be retrieved again because `lookup` performs a left-to-right search that will always encounter the new binding before the old one.

```
function update
  input Env env;
  input Ident id;
  input Value value;
  output Env outEnv;
algorithm
  outEnv := (id, value) :: env;
end update;
```

2.6 AssignTwoType – Introducing Typing

AssignTwoType is an extension of the Assignments language made by introducing `Real` numbers. Now we have two types in the language, `Integer` and `Real`, which creates a need both to check the typing of expressions during evaluation, and to be able to store constant values of two different types in the environment.

2.6.1 Concrete Syntax of AssignTwoType

Real valued constants contain a dot and/or an exponent, as in:

```
3.14159
5.36E-10
11E+5
```

Only one additional rule has been added compared to the BNF grammar of the Assignments language. The non-terminal element can now also expand into a `Real` constant, as shown below:

```
element          :  T_INTCONST
                 |  T_REALCONST
                 |  T_LPAREN  expression  T_RPAREN
```

The lexical specification follows below. One new token type, `T_REALCONST`, has been introduced compared to the Assignments language. The regular expression rcon1 represents a real constant that must contain a dot, whereas `rcon2` must contain an exponent. Any real constant must contain either a dot or an exponent. The ? question mark in the regular expressions signify optional occurrence.

```
/* Lex style lexical syntax of tokens in the language AssignTwoType */
whitespace   [ \t\n]+
letter       [a-zA-Z_]
ident        {letter} ({letter} | {digit})*
digit        [0-9]
digits       {digit}+
icon         {digits}
pt           "."
sign         [+-]
exponent     ([eE]{sign}?{digits})
rcon1        {digits}({pt}{digits}?)?{exponent}
rcon2        {digits}?{pt}{digits}{exponent}?
rcon         {rcon1}|{rcon2}
%%
{whitespace} ;
{ident}      return lex_ident(); /* T_IDENT */
{icon}       return lex_icon();  /* T_INTCONST */
{rcon}       return lex_rcon();  /* T_REALCONST */
":="         return T_ASSIGN;
"+"          return T_ADD;
"-"          return T_SUB;
"*"          return T_MUL;
"/"          return T_DIV;
"("          return T_LPAREN;
")"          return T_RPAREN;
```

2.6.2 Abstract Syntax

The abstract syntax of AssignTwoType has been extended in two ways compared to the Assignments language. A `REAL` node has been inserted into the expression (`Exp`) union type, and a parameterized abstract syntax (Section 2.2) has been selected to enable a more compact semantics part of the specification by grouping cases for similar constructs in the language.

The environment must now be able to store values of two types: `Integer` or `Real`. This is achieved by representing values, of type `Value`, as either `INTval` or `REALval` nodes. We could alternatively have used the `INT` and `REAL` constructors of the `Exp` union type. However, this would have had the disadvantages of mixing up the evaluation value type `Value` with the abstract syntax (which contain many other nodes), and making the strong typing of the specification less orthogonal, thus reducing the probability of the Modelica system catching possible type errors.

An auxiliary union type `Ty2` has been introduced to more conveniently be able to encode the semantics of different combinations of `Integer` and `Real` typed values.

The package header of AssignTwoType precedes the abstract syntax declarations.

```
package AssignTwoType "Assignment language with two types, integer and real"

/* Parameterized abstract syntax for the AssignTwoType language */
uniontype Program
  record PROGRAM  ExpLst expLst;  Exp exp;  end PROGRAM;
end Program;
```

```
uniontype Exp
  record INT    Integer int;                          end INT;
  record REAL   Real real;                            end REAL;
  record BINARY Exp exp1;  BinOp binOp;  Exp exp2;    end BINARY;
  record UNARY  UnOp unOp; Exp exp;                   end UNARY;
  record ASSIGN Ident id;  Exp exp;                   end ASSIGN;
  record IDENT  Ident id;                             end IDENT;
end Exp;

type ExpLst = List<Exp>;

uniontype BinOp
  record ADD  end ADD;
  record SUB  end SUB;
  record MUL  end MUL;
  record DIV  end DIV;
end BinOp;

uniontype UnOp
  record NEG  end NEG;
end UnOp;

type Ident = String;

/* Values, bindings and environments */
uniontype Value "Values stored in environments"
  record INTval  Integer int;  end INTval;
  record REALval  Real real;  end REALval;
end Value;

type VarBnd = Tuple<Ident,Value>;

type Env = List<VarBnd>;

uniontype Ty2  "An auxiliary datatype used to handle types during evaluation"
  record INT2  Integer int1; Integer int2; end INT2;
  record REAL2 Real real1;   Real real2;   end REAL2;
end Ty2;
```

2.6.3 Semantics of AssignTwoType

The semantics of the AssignTwoType language is quite similar to the semantics of the Assignments language described in Section 2.4.4, except for the introduction of multiple types. Having multiple types in a language may give rise to a combinatorial explosion in the number of cases needed, because the semantics of each combination of argument types and operators needs to be described.

In order to somewhat limit this potential "explosion" of cases, we introduce a type lattice (see Section 2.6.3.2), and use the function `type_lub` (for least upper bound of types; Section n order to somewhat limit this potential "explosion" of cases, we introduce a type lattice (see Section 2.6.3.2) which derives the resulting type and inserts possibly needed type conversions. This reduces the number of needed cases for binary operators to two: one for `Integer` results and one for `Real` results. The parameterized abstract syntax makes it possible to place argument evaluation and type handling for binary operators in only those two cases.

2.6.3.1 Expression Evaluation

Compared to the Assignments language, the `eval` function is still quite similar. Values are now tagged by either `INTval` or `REALval`. We have inserted one additional case for `Real` constants, and collected all binary operators together into two cases, and unary operators into two additional cases. The cases for assignments and variable identifiers are the same as before.

We show the application of some cases to a small example, e.g.:

```
44 + 3.14
```

The abstract syntax representation will be:

```
BINARY( INT(44), ADD, REAL(3.14) )
```

On calling `eval`, this will match the case for binary operators and real number results. The first argument will be evaluated to `INTval(44)`, bound to `v1`, and the second argument to `REALval(3.14)` bound to `v2`. The call to `type_lub` will insert a conversion of the first argument from `Integer` to a `Real` value, giving the result `REAL2(44.0, 3.14)`, which also causes `x` to be bound to `44.0` and `y` to be bound to `3.14`. Finally, `applyRealBinop` will apply the operator `ADD` to the two arguments, returning the result `47.14`, which in the form `REALval(47.14)` together with the unchanged environment is the result of the call to function `eval`.

```
function eval
 "Evaluation of an expression inExp in current environment inEnv, returning
  a possibly updated environment outEnv, and an outValue which can be
  either an integer- or real-typed constant value, tagged with constructors
  INTval or REALval, respectively.
    Note: there will be no type error if a real value is assigned to an
  existing integer-typed variable, since the variable will change
  type when it is updated"
  input Env inEnv;
  input Exp inExp;
  output Env outEnv;
  output Value outValue;
algorithm
  (outEnv,outValue) := matchcontinue (inEnv,inExp)
    local
      Env env,env2,env1;
      Integer ival,x,y,z;
      Real rval, x,y,z;
      Value value,v1,v2;
      Ident id;
      Exp e1,e2,e,exp;
      BinOp binop;  UnOp unop;

    case (env,INT(ival)) then (env,INTval(ival));
    case (env,REAL(rval)) then (env,REALval(rval));

    case (env,IDENT(id))                // variable id
      equation
        (env2,value) = lookupextend(env, id);
      then (env2,value);

    case (env,BINARY(e1,binop,e2))      // integer integerBinop integer
      equation
        (env1,v1) = eval(env, e1);
        (env2,v2) = eval(env, e2);
        INT2(x,y) = type_lub(v1, v2);
        z = applyIntBinop(binop, x, y);
      then (env2,INTval(z));

    case (env,BINARY(e1,binop,e2))      // integer/real realBinop integer/real
      equation
        (env1,v1) = eval(env, e1);
        (env2,v2) = eval(env, e2);
        REAL2(xr,yr) = type_lub(v1, v2);
        zr = applyRealBinop(binop, xr, yr);
      then (env2,REALval(z));

    case (env,UNARY(unop,e))            // integerUnop  exp
      equation
        (env1,INTval(x)) = eval(env, e);
        y = applyIntUnop(unop, x);
      then (env1,INTval(y));
```

```
    case (env,UNARY(unop,e))           // real_unop  exp"
      equation
        (env1,REALval(xr)) = eval(env, e);
        yr = applyRealUnop(unop, xr);
      then (env1,REALval(yr));

    // id := exp; eval of an assignment node returns
    // the updated environment and the assigned value.
    case (env,ASSIGN(id,exp))
      equation
        (env1,value) = eval(env, exp);
        env2 = update(env1, id, value);
      then (env2,value);

  end matchcontinue;
end eval;
```

2.6.3.2 Type Lattice and Least Upper Bound

One general way to partially avoid the potential "combinatorial explosion" of semantic cases for different combinations of operators and types is to introduce a type lattice. The trivial type lattice for real and integer (i.e., `Real` and `Integer`) is shown in Figure 2-4 below, using the partial order that `Real` is greater than `Integer` since integers always can be converted to reals, but not the other way around.

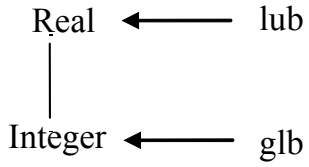

Figure 2-4. Simple type lattice for the types Integer and Real. The least upper bound (lub) is Real; the greatest lower bound (glb) is Integer.

We are however more interested in combinations of two argument types for binary operators, for which the following four cases apply:

- `Real` *op* `Real` => `Real`
- `Real` *op* `Integer` => `Real`
- `Integer` *op* `Real` => `Real`
- `Integer` *op* `Integer` => `Integer`

These cases are represented by the function `type_lub`, introduced below. The function is in fact doing two jobs simultaneously. It is computing the least upper bound of pairs of types, represented by the constructors `INT2` or `REAL2`. Additionally, it performs type conversions of the arguments as needed, to ensure that both arguments become either `Integer` (for `INT2`) or `Real` (for `REAL2`). Thus we will need only two sets of cases for each operator, covering the cases when both arguments are `Integer` or both arguments are `Real`.

```
function type_lub "Type least upper bound, e.g. real and integer gives real"
  input Value inValue1;
  input Value inValue2;
  output Ty2 outTy2;
algorithm
  outTy2 := matchcontinue (inValue1,inValue2)
    local
      Integer x,y;
      Real x2,y2,r,r1,r2;

    case (INTval(x),INTval(y)) then INT2((x,y));

    case (INTval(x),REALval(r))
      equation
        x2 = int_real(x);
```

```
      then REAL2((x2,r));

    case (REALval(r),INTval(y))
      equation
        y2 = int_real(y);
      then REAL2((r,y2));

    case (REALval(r1),REALval(r2)) then REAL2((r1,r2));

  end matchcontinue;
end type_lub;
```

2.6.3.3 Binary and Unary Operators

The essential properties of binary arithmetic operators are described below in the functions `applyIntBinop` and `applyRealBinop`, respectively. Argument evaluation has been taken care of by the two cases for binary operators in the function `eval`, and thus need not be repeated for each case. The type conversion needed for some combinations of `Real` and `Integer` values have already been described by the function `type_lub`, which reduces the number of cases that need to be handled for each operator to two: either `Integer` values (`applyIntBinop`) or `Real` values (`applyRealBinop`).

```
function applyIntBinop "Apply integer binary operator"
  input BinOp inBinop1;
  input Integer inInteger2;
  input Integer inInteger3;
  output Integer outInteger;
algorithm
  outInteger := matchcontinue (inBinop1,inInteger2,inInteger3)
    local Integer x,y;
    case (ADD(),x,y) then x + y;
    case (SUB(),x,y) then x – y;
    case (MUL(),x,y) then x * y;
    case (DIV(),x,y) then x / y;
  end matchcontinue;
end applyIntBinop;

function applyRealBinop "Apply real binary operator"
  input BinOp inBinop1;
  input Real inReal2;
  input Real inReal3;
  output Real outReal;
algorithm
  outReal := matchcontinue (inBinop1,inReal2,inReal3)
    local Real x,y;
    case (ADD(),x,y) then x +. y;
    case (SUB(),x,y) then x +. y;
    case (MUL(),x,y) then x *. y;
    case (DIV(),x,y) then x /. y;
  end matchcontinue;
end applyRealBinop;
```

There is only one unary operator, unary minus, in the current language. Thus the functions `applyIntUnop` and `applyRealUnop` for operations on integer and real values, respectively, become rather short.

```
function applyIntUnop "Apply integer unary operator"
  input UnOp inUnop;
  input Integer inInteger;
  output Integer outInteger;
algorithm
  outInteger := matchcontinue (inUnop,inInteger)
    local Integer x;
    case (NEG(),x) then –x;
  end matchcontinue;
end applyIntUnop;
```

```
function applyRealUnop "Apply real unary operator"
  input UnOp inUnop;
  input Real inReal;
  output Real outReal;
algorithm
  outReal := matchcontinue (inUnop,inReal)
    local Real x;
    case (NEG(),x) then -.x;
  end matchcontinue;
end applyRealUnop;
```

2.6.3.4 Functions for Lookup and Environment Update

We give the usual functions for lookup and environment update. Stored values may be either integers, tagged by `INTval()`, or real numbers tagged by `REALval()`. However, there is no declaration of types or static typing of variables in this language. A variable gets its type when it is assigned a value.

```
function lookup  "lookup returns the value associated with an identifier.
                  If no association is present, lookup will fail."
  input Env env;
  input Ident id;
  output Value outValue;
algorithm
  outValue :=
  matchcontinue (env,id)
    local  Ident id2,id;  Value value;  Env rest;
    case ((id2,value) :: rest, id) then
      if id==id2 then value   // id first in list
      else lookup(rest,id);   // id in rest of list
  end matchcontinue;
end lookup;


function lookupextend
 "lookupextend returns the value associated with an identifier and an updated
  environment. If no association is present, lookupextend will fail."
  input Env inEnv;
  input Ident inIdent;
  output Env outEnv;
  output Value outValue;
algorithm
  (outEnv,outValue) := matchcontinue (inEnv,inIdent)
    local
      Value value;  Env env;  Ident id;

    // Return value of id in env.
    // If id not present, add id and return 0
    case (env,id)
      equation
        failure(v = lookup(env, id));
        value = INTval(0);
      then ((id,value) :: env,value);

    case (env,id)
      equation
        value = lookup(env, id);
      then (env,value);

  end matchcontinue;
end lookupextend;

function update  "update returns an updated environment with a new
                  (id,value) association"
  input Env env;
  input Ident id;
  input Value value;
  output Env outEnv;
```

```
algorithm
  outEnv := (id,value) :: env;
end update;

end AssignTwoType;
```

2.7 A Modular Specification of the PAMDECL Language

PAMDECL is PAM extended with declarations of variables and two types: `Integer` and `Real`. Thus it combines the properties of both PAM and AssignTwoType. The specification is modular, including separate packages for different aspects.

In general, Modelica packages facilitates writing modular specifications, where each package describes some related aspects of the specified language. Thus, it is common to specify the abstract syntax in a special package and other aspects such as evaluation, translation, or type elaboration in separate packages.

We present a modularized version of the complete abstract syntax and semantics for PamDecl in Appendix C.2, using five packages: `Main` for the main program, `ScanParse` for scanning and parsing, `Absyn` for abstract syntax, `Env` for variable bindings and environment handling, and `Eval` for evaluation.

A package must import definitions from other packages in order to reference items declared in those packages.

References to names defined in other packages must be prefixed by the defining package name followed by a dot, as in `Absyn.ASSIGN()`, see Section 11.2.1, when referencing the `ASSIGN` constructor from the package `Absyn`.

2.8 Summary

In this chapter we present a series of small example languages to introduce MetaModelica together with techniques for programming language specification. We start with the very simple Exp1 language, containing simple integer arithmetic and integer constants. Then follows a short section on the parameterized style of abstract syntax. The Exp2 specification describes the same language as Exp1 but shows the consequences of using parameterized abstract syntax. The Assignments language extends Exp1 with variables and assignments, thus introducing the concept of environment.

The small Pascal-like PAM language further extends our toy language by introducing control structures such as if-then-else statements, loops (but not goto), and simple input/input. However, PAM does not include procedures and multiple variable types. Only integer variables are handled by the produced evaluator. PAM also introduces relational expressions. Parameterized abstract syntax is used in the specification.

Our next language, called AssignTwoType, is designed to introduce multiple variable types in the language. It is the same language as Assignments, but adding real values and variables, and employing the parameterized style of abstract syntax. The concept of type lattice is also introduced in this section.

Next, we present the concept of Modelica packages, to show how different aspects of a specification such as abstract syntax, environment handling, evaluation cases, etc. can be separated into different packages. Such modularization is especially important for large specifications.

Finally, we combine the constructs of the PAM language, the multiple variable types of AssignTwoType and the usage of Modelica packages, to produce a modular specification of a language called PAMDECL, which is PAM extended with declarations and multiple (integer and real) variable types.

The style of all specifications so far has been “evaluative” in nature, aiming at producing interpreters. In Chapter 3 we will present “translational” style specifications, from which compilers can be generated.

2.9 Exercises

A number of exercises concerning some MetaModelica language constructs and interpretive semantics modeling of small languages are available in Appendix D. Solutions are available in Appendix E. Translational semantics exercises can be found in Chapter 3. Exercises for functional programming and higher order functions are also available in Appendix D..

Chapter 3

Translational Semantics

A compiler is a translator from a source language to a target language. Thus, it would be rather natural if the idea of translation is somehow reflected in the semantic definition of a programming language. In fact, the meaning of a programming language can be precisely described by defining the meaning (semantics) of the source language in terms of a translation to some target (object) language, together with a definition of the semantics of the object language itself, see Figure 3-1. This is called a translational semantics of the programming language.

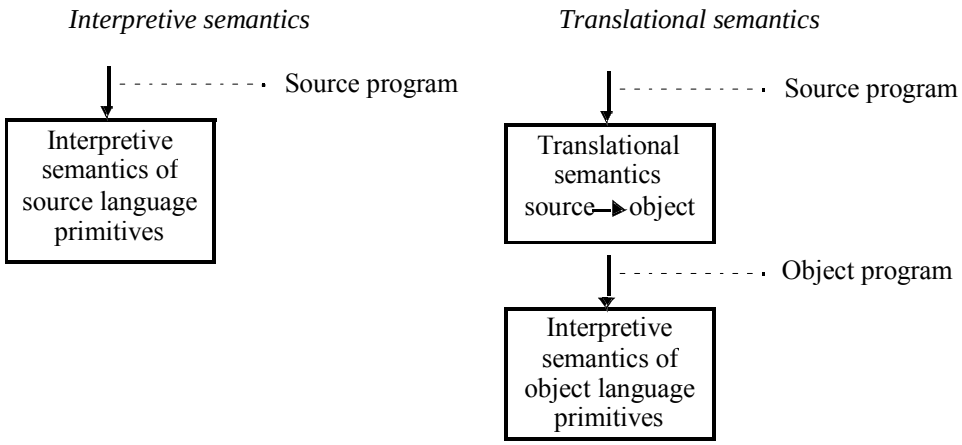

Figure 3-1. A comparison between an interpretive semantics and translational semantics. In an interpretive semantics, the computational meaning of source language primitives are directly defined, e.g. using MetaModelica. In a translational semantics, the meaning is defined as a translation to object language primitives, which in turn are defined using an interpretive semantics.

However, so far in this text we have primarily focused on how to define the semantics of programming languages directly in terms of evaluation of MetaModelica primitives. That style of semantics specification, called interpretive semantics, can be used for automatic generation of interpreters which interpret abstract syntax representations of source programs. Analogously, a translational semantics can be used for the generation of a compiler from a source language to a target language, as briefly mentioned in Section 1.4.

There are also techniques based on partial evaluation (Jones, Gomard, and Sestoft 1993), for the generation of compilers from certain styles of interpretive semantics. However, these techniques often give unpredictable results and performance problems. Therefore, in the rest of this text we will exclusively use translational semantics as a basis for practical compiler generation.

In fact, writing translational semantics is usually not harder than writing interpretive semantics. One just has to keep in mind that the semantics is described in two parts: the meaning of source language primitives in terms of (a translation to) target language primitives, and the meaning of the target

primitives themselves. A simplified picture of compiler generation from translational semantics is shown in Figure 3-2.

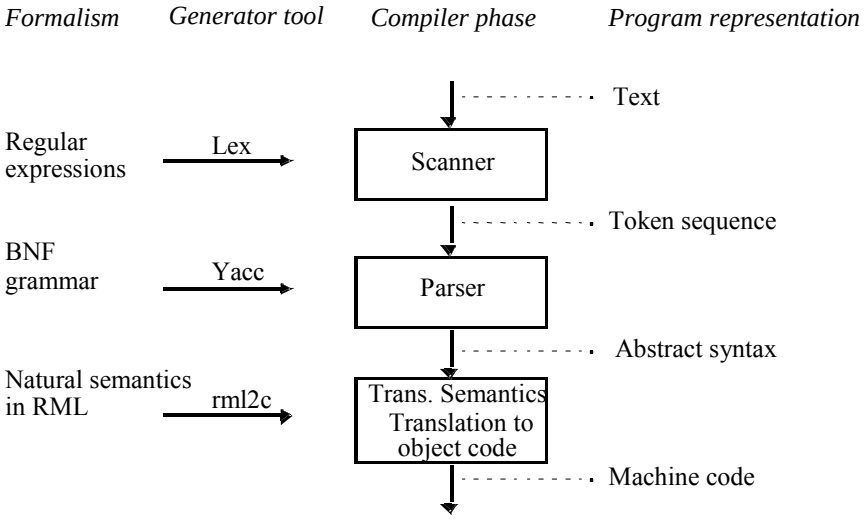

Figure 3-2. Simplified version of compiler generation based on translational semantics. The semantics of a language is specified directly in terms of object code primitives. In comparison to Figure 1-1, the optimization and final code generation phases have been excluded.

3.1 Translating PAM to Machine Code

As an introduction translational semantics, we will specify the translational semantics of a simple language, with the goal of generating a compiler from this language to machine code. The simple PAM language has already been described, and an interpretive semantics has been given in Section 2.5. This makes it a natural first choice for a translational semantics. In Chapter 3 of (Pagan 1981), an attribute grammar style translational semantics of PAM can be found. It is instructive to compare the attribute grammar specification to the MetaModelica style translational semantics of PAM described in this chapter. The target assembly language described in the next section has been chosen to be the same as in (Pagan 1981) to simplify parallel study.

3.1.1 A Target Assembly Language

In the translational approach, a target language for the translation process is needed. Here we choose a very simple assembly (machine code) language, which is similar to realistic assembly languages, but very much simplified. For example, this machine has only one register (an accumulator) and much fewer instructions than commercial microprocessors. Still, it is complete enough to reflect most properties of realistic assembly languages. There are 17 types of instructions, listed below:

```
LOAD       Load accumulator
STO        Store
ADD        Add
SUB        Subtract
MULT       Multiply
DIV        Divide
GET        Input a value
PUT        Output a value
J          Jump
JN         Jump on negative
JP         Jump on positive
JNZ        Jump on negative or zero
JPZ        Jump on positive or zero
JNP        Jump on negative or positive
```

```
LAB        Label (no operation)
HALT       Halt execution
```

All instructions, except `HALT`, have one operand. For example, `LOAD X`, will load the variable at address `X` into the accumulator. Conversely, `STO X` will store the current value in the accumulator at the address specified by `X`. The instructions `ADD`, `SUB`, `MULT`, and `DIV` perform arithmetic operations on two values, the accumulator value and the operand value. Operands can be integer constants or symbolic addresses of variables or temporaries (`T1`,`T2`,...), or symbolic labels representing code addresses. Instructions which compute a result always store it in the accumulator. For example, `SUB X` means that accumulator-`X` is computed, and stored in the accumulator.

The input/output instructions `GET X` and `PUT X` will input and output a value to variable `X`, respectively. There are 5 conditional jump instructions and one unconditional jump. The conditional jumps are: `JN`,`JP`,`JNZ`,`JPZ`, and `JNP` which jump to a label (address) conditionally on the current value in the accumulator. The `J L1` instruction is an example of an unconditional jump to the label `L1`. The `LAB` pseudo instruction is no instruction, it just declares the position of a label in the code. Finally, the `HALT` instruction stops execution.

3.1.2 A Translated PAM Example Program

Before going into the details of the translational semantics, it is instructive to take a look at the translation of a small PAM example PAM program, shown below:

```
read x,y;
while x <> 99 do
  ans :=  (x + 1) - (y / 2);
  write ans;
  read x,y;
end
```

This example program is translated into the following assembly code, presented in its textual representation:

```
GET   x                    STO   T1
GET   y                    LOAD  T0
L2    LAB                  SUB   T1
LOAD  x                    STO   ans
SUB   99                   PUT   ans
JZ    L3                   GET   x
LOAD  x                    GET   y
ADD   1                    J     L2
STO   T0                   L3    LAB
LOAD  y                    HALT
DIV   2
```

However, to simplify and structure the translational semantics of PAM, the target language will be a structured representation of the assembly code, called Mcode, which is defined in MetaModelica. The Mcode representation of the translated program, as shown below, is finally converted into the textual representation previously presented.

All Mcode operators start with the letter `M`. Binary arithmetic operators are grouped under the node `MB`, and conditional jump operators under `MJ`. There are four kinds of operands, indicated by the constructors `I` (Identifier), `L` (Label), `N` (Numeric integer), and `T` (for Temporary).

```
MGET  ( I(x) )                 MSTO  ( T(2) )
MGET  ( I(y) )                 MLOAD ( T(1) )
MLABEL( L(1) )                 MB(MSUB,T(2) )
MLOAD ( I(x) )                 MSTO  ( I(ans) )
MB(MSUB,N(99))                 MPUT  ( I(ans) )
MJ(MJZ, L(2) )                 MGET  ( I(x) )
MLOAD ( I(x) )                 MGET  ( I(y) )
MB(MADD,N(1) )                 MJMP  ( L(1) )
MSTO  ( T(1) )                 MLABEL( L(2) )
```

```
MLOAD ( I(y) )                          MHALT
MB(MDIV,N(2) )
```

3.1.3 Abstract Syntax for Machine Code Intermediate Form

The abstract syntax of the structured machine code representation, called Mcode, is defined in MetaModelica below. We group the four arithmetic binary operators `MADD`, `MSUB`, `MMULT` and `MDIV` in the union type `MBinOp`. The six conditional jump instructions `MJMP`,`MJP`,`MJN`,`MJNZ`,`MJPZ`,`MJZ` are represented by constructors in the union type `MCondJmp`. As usual, this grouping of similar constructs simplifies the semantic description. There are four kinds of operands: identifiers, numeric constants, labels, and temporaries. For these we have defined the type aliases `MLab`, `MTemp`, `MIdent`, `MidTemp` in order to make the translational semantics more readable.

The constructors `MB` and `MJ` are used for binary arithmetic instructions and conditional jumps, respectively. The first argument to these constructors indicates the specific arithmetic operation or conditional jump.

```
package Mcode

uniontype MBinOp
  record MADD  end MADD;
  record MSUB  end MSUB;
  record MMULT end MMULT;
  record MDIV  end MDIV;
end MBinOp;

uniontype MCondJmp
  record MJNP  end MJNP;
  record MJP   end MJP;
  record MJN   end MJN;
  record MJNZ  end MJNZ;
  record MJPZ  end MJPZ;
  record MJZ   end MJZ;
end MCondJmp;

uniontype MOperand
  record I  Id id;        end I;
  record N  Integer int;  end N;
  record T  Integer int;  end T;
end MOperand;

type MLab    = MOperand; // Label
type MTemp   = MOperand; // Temporary
type MIdent  = MOperand; // Identifier
type MIdTemp = MOperand; // Id or Temporary

uniontype MCode
  record MB     MBinOp mBinOp;  Moperand Moperand; end MB; // Binary arith ops
  record MJ     MCondJmp mCondJmp; MLab mLab;      end MJ; // Conditional jumps
  record MJMP   Mlab mlab;                         end MJMP;
  record MLOAD  MIdTemp mIdTemp;                   end MLOAD;
  record MSTO   MIdTemp mIdTemp;                   end MSTO;
  record MGET   MIdent mIdent;                     end MGET;
  record MPUT   MIdent mIdent;                     end MPUT;
  record MLABEL MLab mLab;                         end MLABEL;
  record MHALT                                     end MHALT;
end MCode;

end Mcode;
```

3.1.4 Concrete Syntax of PAM

The concrete syntax of PAM has already been described in Section 2.5.2.

3.1.5 Abstract Syntax of PAM

The abstract syntax of PAM is identical to that described in Section 2.5.3. It is repeated here for convenience.

```
package Absyn  "Parameterized abstract syntax for the PAM language"

type Ident = String;

uniontype BinOp
  record ADD  end ADD;
  record SUB  end SUB;
  record MUL  end MUL;
  record DIV  end DIV;
end BinOp;

uniontype RelOp
  record EQ  end EQ;
  record GT  end GT;
  record LT  end LT;
  record LE  end LE;
  record GE  end GE;
  record NE  end NE;
end RelOp;

uniontype Exp
  record INT      Integer int;  end INT;
  record IDENT    Ident   id;   end IDENT;
  record BINARY   Exp exp1; BinOp op; Exp exp2;  end BINARY;
  record RELATION Exp exp1; RelOp op; Exp exp2;  end RELATION;
end Exp;

type IdentList = List<Ident>;
uniontype Stmt
  record ASSIGN Ident id; Exp exp; end ASSIGN;           // Id := Exp
  record IF     Exp exp; Stmt stmt1; Stmt stmt2; end IF; // if Exp then Stmt ...
  record WHILE  Exp exp; Stmt stmt;  end WHILE;          // while Exp do Stmt"
  record TODO   Exp exp; Stmt stmt;  end TODO;           // to Exp do Stmt..."
  record READ   IdentList  idlist;  end READ;           // read id1,id2,..."
  record WRITE  IdentList  idlist;  end WRITE;          // write id1,id2,.."
  record SEQ    Stmt stmt1;  Stmt stmt2;  end SEQ;      // Stmt1; Stmt2"
  record SKIP  end SKIP;                                // ; empty stmt"
end Stmt;

end Absyn;
```

3.1.6 Translational Semantics of PAM

The translational semantics of PAM consists of several separate parts. First we describe the translation of arithmetic expressions, which is the simplest case. Then we turn to comparison expressions which occur in the conditional part of if-statements and while-statements. Such comparisons are translated into conditional jump instructions. Next, the translation of all statement types in PAM are described together with the translation of a whole program. Finally, a MetaModelica program for emitting assembly text from the structured MCode representation is presented, although this is not really part of the translational semantics of PAM.

3.1.6.1 Arithmetic Expression Translation

The translation of binary arithmetic expressions is specified by the `transExpr` function together with two small help functions `transBinop` and `gentemp`. The `transBinop` function just translates the four arithmetic node types in the abstract syntax into corresponding `MCode` node types. Each call to the gentemp generator function produces a unique label of type `L1`, `L2`, etc.

The `transExpr` function contains essentially all semantics of PAM arithmetic expressions. The first two cases handle the simple cases of expressions which are either an integer constant or a variable. The generated code is in the form of a list of `MCode` tuples, as is reflected in the signature of the `transExpr` function below:

```
function transExpr "Arithmetic expression translation"
  input Absyn.Exp inExp;
  output List<Mcode.MCode> outMCodeList;
algorithm
  ...
    case Absyn.INT(v) then List(Mcode.MLOAD(Mcode.N(v)));  // integer constant
    case Absyn.IDENT(id) then List(Mcode.MLOAD(Mcode.I(id))); // identifier id
```

The semantics of computing a constant or a variable is to load the value into the accumulator, as in the following instruction where `id` is the variable `X4`:

```
MLOAD( I(X4) )
```

and in assembly text form:

```
LOAD   X4
```

The first case is for simple binary arithmetic expressions such as `e1 – e2` where expression `e2` only is a constant or a variable which gives rise to a load instruction (see the second local equation in the case). The code for this expression is as follows, where `MB` denotes a binary operator and `MSUB` subtraction:

```
<code for expression e1>
MB(MSUB(),   e2)
```

and in assembly text form:

```
<code for expression e1>
SUB    e2
```

The corresponding case follows below.

```
// Arith binop: simple case, expr2 is just an
// identifier or constant: expr1 binop expr2
case Absyn.BINARY(e1,binop,e2)
  equation
    cod1 = transExpr(e1);
    List(Mcode.MLOAD(operand2)) = transExpr(e2);  // Condition expr2 simple
    opcode = transBinop(binop);
    cod3 = listAppend(cod1, List(Mcode.MB(opcode,operand2)));
  then cod3;
```

The second case handles binary arithmetic expressions such as `e1-e2`, `e1+e2`, etc., where `e2` can be a complicated expression. The code pattern for `e1-e2` in assembly text form becomes:

```
<code for e1>
STO    T1

<code for e2>
STO    T2
LOAD   T1
SUB    T2
```

The case is presented below. The generated code for expressions `e1` and `e2` are bound to `cod1` and `cod2`, respectively. The binary operation is translated to the MCode version, which is bound to `opcode`. Then two temporaries are produced. Finally a code sequence is produced which closely follows the code pattern above. The function `listAppend6` appends the elements of six argument lists, whereas the standard `listAppend` only accepts two list arguments.

```
// Arith binop: general case, expr2 is a more
// complicated expr: expr1 binop expr2
case Absyn.BINARY(e1,binop,e2)
  equation
    cod1 = transExpr(e1);
```

```
      cod2 = transExpr(e2);
      opcode = transBinop(binop);
      t1 = gentemp();
      t2 = gentemp();
      cod3 = listAppend6(cod1, // code for expr1
        {Mcode.MSTO(t1)},       // store expr1
        cod2,                   // code for expr2
        {Mcode.MSTO(t2)},       // store expr2
        (Mcode.MLOAD(t1)},      // load expr1 value into Acc
        {Mcode.MB((opcode,t2))} // Do arith operation
      );
    then cod3;
```

As one additional example, we show the following expression:

```
(x + y * z) + b * c
```

This is translated into the following code sequence:

```
LOAD   x            STO    T3
STO    T1           LOAD   b
LOAD   y            MULT   c
MULT   z            STO    T4
STO    T2           LOAD   T3
LOAD   T1           ADD    T4
ADD    T2
```

Note that the two cases for binary arithmetic operations overlap. The first case covers the simple case where the second expression is just an identifier or constant, and will give rise to more compact code than the second case which covers both the simple and the general case. From a semantic point of view, the first case is not needed since the second case specifies the same semantics for simple arithmetic expressions as the second case, even though the second case will give rise to more instructions in the translated code. Still, it is not incorrect to keep the first case, since the PAM semantics is not changed by it.

Operationally, MetaModelica will evaluate the cases in top-down order, and thus will use the more specific first case whenever it matches. Therefore we keep the first case in order to obtain a compiler that produces slightly more efficient code than otherwise possible.

The complete `transExpr` function follows below, together with some help functions:

```
function transExpr "Arithmetic expression translation"
  input Absyn.Exp inExp;
  output List<Mcode.MCode> outMCodeList;
algorithm
  outMCodeList := matchcontinue (inExp)
    local
      Integer v;
      String id;
      MCodeList cod1,cod3,cod2;
      Mcode.MOperand operand2,t1,t2;
      Mcode.MBinOp opcode;
      Absyn.Exp e1,e2;
      Absyn.BinOp binop;

    case Absyn.INT(v) then List(Mcode.MLOAD(Mcode.N(v)));  // integer constant
    case Absyn.IDENT(id) then List(Mcode.MLOAD(Mcode.I(id))); // identifier id

    // Arith binop: simple case, expr2 is just an
    // identifier or constant:  expr1 binop expr2
    case Absyn.BINARY(e1,binop,e2)
      equation
        cod1 = transExpr(e1);
        List(Mcode.MLOAD(operand2)) = transExpr(e2);
        opcode = transBinop(binop); // expr2 simple
        cod3 = listAppend(cod1, List(Mcode.MB(opcode,operand2)));
      then cod3;
```

```
    // Arith binop: general case, expr2 is a more
    // complicated expr:   expr1 binop expr2
    case Absyn.BINARY(e1,binop,e2)
      equation
        cod1 = transExpr(e1);
        cod2 = transExpr(e2);
        opcode = transBinop(binop);
        t1 = gentemp();
        t2 = gentemp();
        cod3 = listAppend6(cod1,  // code for expr1
          {Mcode.MSTO(t1)},       // store expr1
          cod2,                   // code for expr2
          {Mcode.MSTO(t2)},       // store expr2
          (Mcode.MLOAD(t1)},      // load expr1 value into Acc
          {Mcode.MB(opcode,t2)}   // Do arith operation
        );
    then cod3;
  end matchcontinue;
end transExpr;

function transBinop  "Translate binary operator from Absyn to MCode"
  input Absyn.BinOp inBinop;
  output Mcode.MBinOp outMBinop;
algorithm
  outMBinop := match (inBinop)
    case Absyn.ADD() then Mcode.MADD();
    case Absyn.SUB() then Mcode.MSUB();
    case Absyn.MUL() then Mcode.MMULT();
    case Absyn.DIV() then Mcode.MDIV();
  end match;
end transBinop;

function gentemp  "Generate temporary"
  output Mcode.MOperand outMOperand;
protected
  Integer no;
algorithm
  no = tick();
  outMOperand := Mcode.T(no);
end gentemp;

function listAppend6<Type_a>
  input List<Type_a> l1;
  input List<Type_a> l2;
  input List<Type_a> l3;
  input List<Type_a> l4;
  input List<Type_a> l5;
  input List<Type_a> l6;
  output List<Type_a> l16;
protected
  List<Type_a> l13,l46;
algorithm
  l13 = listAppend3(l1, l2, l3);
  l46 = listAppend3(l4, l5, l6);
  l16 = listAppend(l13, l46);
end listAppend6;
```

3.1.6.2 Translation of Comparison Expressions

Comparison expressions have the form <*expression*><*relop*> <*expression*>, as for example in:

```
x <  5
y >= z
```

In the simple PAM language, such comparison expressions only occur as predicates in if-statements and while-statements. If the predicate is true, then the body of the if-statement should be executed, otherwise

jump over it to some label if the predicate is false. Thus, a conditional jump to a label occurs if the predicate is false.

This is reflected in the translation of relational operators by the function `transRelop`, where the selected conditional jump instruction is logically opposite to the relational operator. For example, regarding the comparison x <= y which is equivalent to x - y <= 0 if we ignore the fact that overflow or underflow of arithmetic operations can cause errors, a jump should occur if the comparison is false, i.e., x - y > 0, meaning that the relational operator `LE` (less or equal) should be translated to `MJP` (jump on positive):

```
function transRelop "Translate relation operator"
  input Absyn.RelOp inRelop;
  output Mcode.MCondJmp outMCondJmp;
algorithm
  outMCondJmp := match inRelop
    case Absyn.EQ() then Mcode.MJNP(); // Jump on Negative or Positive
    case Absyn.LE() then Mcode.MJP();  // Jump on Positive
    case Absyn.LT() then Mcode.MJPZ(); // Jump on Positive or Zero
    case Absyn.GT() then Mcode.MJNZ(); // Jump on Negative or Zero
    case Absyn.GE() then Mcode.MJN();  // Jump on Negative
    case Absyn.NE() then Mcode.MJZ();  // Jump on Zero
  end match;
end transRelop;
```

Translation of the actual comparison expression is described by the `transComparison` function, having the following signature:

```
function transComparison "Translate comparison relation operator"
  type MCodeList = List<Mcode.MCode>;
  input Absyn.Comparison inComparison;
  input Mcode.MLab inMLab;
  output MCodeList outMCodeList;
```

The label argument is needed as an argument to the generated conditional jump instruction. The following code sequence is suitable for all comparison expressions having the structure `e1` <*relop*> `e2`, here represented by the example `e1 <= e2`, which is equivalent to `0 <= e2-e1`:

```
<code  for e1>
STO    T1
<code for e2>
SUB    T1           /* Compute e2-e1; comparison false if negative */
JN     Lab          /* Jump to label Lab if negative */
```

The second case in the `transComparison` function translates according to this pattern, as shown below. The first case applies to the special case when `e2` is a variable or a constant, and can then avoid using a temporary.

```
case (Absyn.RELATION(e1,relop,e2),lab)  // expr1 relop expr2
  equation
    cod1 = transExpr(e1);
    List(Mcode.MLOAD(operand2)) = transExpr(e2);
    jmpop = transRelop(relop);
    cod3 = listAppend3(cod1, {Mcode.MB(Mcode.MSUB(),operand2)},
                       {Mcode.MJ(jmpop,lab)} );
  then cod3;
```

The functions needed for translation of comparison expressions, including `transComparison`, follow below:

```
/*************** Comparison expression translation *************/

function transComparison
  "Translation of a comparison: expr1 relop expr2
   Example call:  transComparison(RELATION(INDENT(x), GT, INT(5)), L(10))"
  input Absyn.Comparison inComparison;
  input Mcode.MLab inMLab;
  output List<Mcode.MCode> outMCodeList;
```

```
algorithm
  outMCodeList := matchcontinue (inComparison,inMLab)
    local
      List<Mcode.MCode> cod1,cod3,cod2;
      Mcode.MOperand operand2,lab,t1;
      Mcode.MCondJmp jmpop;
      Absyn.Exp e1,e2;
      Absyn.RelOp relop;
   /*
    * Use a simple code pattern (the first case), when expr2 is  a simple
    * identifier or constant:
    *   code for expr1
    *   SUB operand2
    *   conditional jump to lab
    *
    * or a general code pattern (second case), which is needed when expr2
    * is more complicated than a simple identifier or constant:
    *   code for expr1
    *   STO temp1
    *   code for expr2
    *   SUB temp1
    *   conditional jump to lab
    */
    case (Absyn.RELATION(e1,relop,e2),lab)   // Simple case, expr1 relop expr2
      equation
        cod1 = transExpr(e1);
        List(Mcode.MLOAD(operand2)) = transExpr(e2);  // Simple if a load
        jmpop = transRelop(relop);
        cod3 = listAppend3(cod1, {Mcode.MB(Mcode.MSUB(),operand2)},
                           {Mcode.MJ(jmpop,lab)} );
      then cod3;

    case (Absyn.RELATION(e1,relop,e2),lab) // Complicated, expr1 relop expr2
      equation
        cod1 = transExpr(e1);
        cod2 = transExpr(e2);
        jmpop = transRelop(relop);
        t1 = gentemp();
        cod3 = listAppend5(cod1, {Mcode.MSTO(t1)}, cod2,
          {Mcode.MB(Mcode.MSUB(),t1)}, {Mcode.MJ(jmpop,lab)} );
      then cod3;

  end matchcontinue;
end transComparison;

function transRelop  "Translate comparison relation operator.
  Note that for these relational operators, the selected jump
  instruction is logically opposite. For example, if equality to zero
  is true, we should should just continue, otherwise jump (MJNP)"
  input Absyn.RelOp inRelop;
  output Mcode.MCondJmp outMCondJmp;
algorithm
  outMCondJmp := match (inRelop)
    case Absyn.EQ() then Mcode.MJNP(); // Jump on Negative or Positive
    case Absyn.LE() then Mcode.MJP();  // Jump on Positive
    case Absyn.LT() then Mcode.MJPZ(); // Jump on Positive or Zero
    case Absyn.GT() then Mcode.MJNZ(); // Jump on Negative or Zero
    case Absyn.GE() then Mcode.MJN();  // Jump on Negative
    case Absyn.NE() then Mcode.MJZ();  // Jump on Zero
  end match;
end transRelop;
```

3.1.6.3 Statement Translation

We now turn to the translational semantics of the different statement types of PAM, which is described by the cases of the function `transStmt`.

The first case specifies translation of an assignment statement `id := e1;` which is particularly simple. Just compute the value of `e1` and store in variable `id`, according to the following code pattern:

```
<code for e1>
STO   id
```

and the case:

```
// Assignment Statement translation: map the current state into a new state
case Absyn.ASSIGN(id,e1)
  equation
    cod1 = transExpr(e1);
    cod2 = listAppend(cod1, {Mcode.MSTO(Mcode.I(id))} );
  then cod2;
```

Translation of an empty statement, represented as a `SKIP` node, is very simple since only an empty instruction sequence is produced as in the case below:

```
case Absyn.SKIP  then {};  // ; empty statement
```

Translation of if-statements is more complicated. There are two cases, the first valid for if-then statements in the form if comparison then `s1` using the code pattern:

```
<code for comparison with conditional jump to L1>
<code for s1>
LABEL L1
```

and the case:

```
case Absyn.IF(comp,s1,Absyn.SKIP)                /* if comp then s1 */
  equation
    s1cod = transStmt(s1);
    l1 = genlabel();
    compcod = transComparison(comp, l1);
    cod3 = listAppend3(compcod, s1cod, {Mcode.MLABEL(l1)} );
  then cod3;
```

Note that if-then statements are represented as if-then-else statement nodes with an empty statement (`SKIP`) in the else-part.

General if-then-else statements of the form **`if`** `comparison` **`then`** `s1` **`else`** `s2` are using the code pattern:

```
<code for comparison with conditional jump to L1>
<code for s1>
J     L2
LABEL L1
<code for s2>
LABEL L2
```

and the case:

```
case Absyn.IF(comp,s1,s2)             /* if comp then s1 else s2 */
  equation
    s1cod = transStmt(s1);
    s2cod = transStmt(s2);
    l1 = genlabel();
    l2 = genlabel();
    compcod = transComparison(comp, l1);
    cod3 = listAppend6(
       compcod, s1cod,
       {Mcode.MJMP(l2)},
       {Mcode.MLABEL(l1)},
       s2cod,
       {Mcode.MLABEL(l2)} );
  then cod3;
```

This second case also specifies correct semantics for if-then statements, although one unnecessary jump instruction would be produced. Avoiding this jump is the only reason for keeping the first case.

We now turn to while-statements of the form **`while`** `comparison` **`do`** `s1`. This is an iterative statement and thus contain a backward jump in its code-pattern below:

```
LABEL  L1
<code for comparison, including conditional jump to L2>
<code for s1>
J      L1
LABEL  L2
```

with the case:

```
case Absyn.WHILE(comp,s1)          // while ...
  equation
    bodycod = transStmt(s1);
    l1 = genlabel();
    l2 = genlabel();
    compcod = transComparison(comp, l2);
    cod3 = listAppend5(
      {Mcode.MLABEL(l1)},
      compcod, bodycod,
      {Mcode.MJMP(l1)},
      {Mcode.MLABEL(l2)} );
  then cod3;
```

The definite loop statement of the form **`to`** `e1` **`do`** `s1` is a kind of for-statement that found in many other languages. The statement `s1` is executed the number of times specified by evaluating expression `e1` once at the beginning of its execution. The value of `e1` initializes a temporary counter variable which is decremented before each iteration. The loop is exit when the counter becomes negative. The code pattern follows below:

```
<code for e1>
STO    T1           /* T1 is the counter */
LABEL  L1
LOAD   T1
SUB    1            /* Decrement T1 */
JN     L2           /* Exit the loop */
STO    T1
<code for s1>
J      L1
LABEL  L2
```

and the case:

```
case Absyn.TODO(e1,s1)             // to e1 do s1 ...
  equation
    tocod = transExpr(e1);
    bodycod = transStmt(s1);
    t1 = gentemp();
    l1 = genlabel();
    l2 = genlabel();
    cod3 = listAppend10(
      tocod,
      {Mcode.MSTO(t1)},
      {Mcode.MLABEL(l1)},
      {Mcode.MLOAD(t1)},
      {Mcode.MB(Mcode.MSUB(),Mcode.N(1))},
      {Mcode.MJ(Mcode.MJN,l2)},
      {Mcode.MSTO(t1)},
      bodycod,
      {Mcode.MJMP(l1)},
      {Mcode.MLABEL(l2)} );
  then cod3;
```

Next we turn to the input/output statements of PAM. A read-statement of the form `read id1,id2,id3...` will input values to the variables `id1, id2, id3` etc. in that order. This is accomplished by generating code according to the following pattern:

```
GET    id1
GET    id2
GET    id3
...
```

The translation is specified by the first case, stating that reading an empty list of variables produces an empty sequence of `GET` instructions, whereas the second case specifies emission of one `GET` instruction for the first identifier in the non-empty list, and then recursively invokes `transStmt` for the rest of the identifiers in the list. The cases follow below:

```
case Absyn.READ({})  then {};          // read {}

case Absyn.READ(id :: idListRest)      // read id1,id2,...
  equation
    cod2 = transStmt(Absyn.READ(idListRest));
  then Mcode.MGET(Mcode.I(id) :: cod2);
```

The translation of write-statements of form `write id1,id2,id3`,... is analogous to that of read-statements, but produces `PUT` instructions as in:

```
PUT    id1
PUT    id2
PUT    id3
...
```

The translation is specified by the following two cases:

```
case Absyn.WRITE({}) then {};              // write {}

case Absyn.WRITE(id :: idListRest)         // write id1,id2,...
  equation
    cod2 = transStmt(Absyn.WRITE(idListRest));
  then Mcode.MPUT(Mcode.I(id) :: cod2);
```

A sequence of two statements, of the form `stmt1; stmt2` is represented by the abstract syntax node `SEQ`. Since one or both statements can be a statement sequence itself, sequences of arbitrary length can be represented. The instructions from translating two statements in a sequence are simply concatenated as in the case below:

```
case Absyn.SEQ(stmt1,stmt2)                 // stmt1 ; stmt2
  equation
    cod1 = transStmt(stmt1);
    cod2 = transStmt(stmt2);
    cod3 = listAppend(cod1, cod2);
  then cod3;
```

The semantics of translating a whole PAM program is described by a translation of the program body, which is a statement, followed by the `HALT` instruction. This is clear from the function `transProgram` below:

```
function transProgram   "Translate a whole program"
  type MCodeList = List<Mcode.MCode>;
  input Absyn.Stmt progbody;
  output MCodeList programcode;
protected
  MCodeList cod1;
algorithm
  cod1 := transStmt(progbody);
  programcode := listAppend(cod1, {Mcode.MHALT()});
end transProgram;
```

Finally, the complete translational semantics of PAM statements is presented below as the cases and cases of the function `transStmt`.

```
//*************** Statement translation *************
```

```
function transStmt "Statement translation"
  input  Absyn.Stmt        inStmt;
  output List<Mcode.MCode> outMCodeList;
algorithm
  outMCodeList := matchcontinue (inStmt)
    local
      List<Mcode.MCode> cod1,cod2,s1cod,compcod,cod3,s2cod,bodycod,tocod;
      String id;
      Absyn.Exp e1,comp;
      Mcode.MOperand l1,l2,t1;
      Absyn.Stmt s1,s2,stmt1,stmt2;
      List<String> idListRest;

    // Assignment Statement translation: map the current state into a new state
    case Absyn.ASSIGN(id,e1)
      equation
        cod1 = transExpr(e1);
        cod2 = listAppend( cod1, {Mcode.MSTO(Mcode.I(id))} );
      then cod2;

    // ; empty statement
    case Absyn.SKIP then {};

    // if comp then s1
    case Absyn.IF(comp,s1,Absyn.SKIP)
      equation
        s1cod = transStmt(s1);
        l1 = genlabel();
        compcod = transComparison(comp, l1);
        cod3 = listAppend3( compcod, s1cod, {Mcode.MLABEL(l1)} );
      then cod3;

    // if comp then s1 else s2
    case Absyn.IF(comp,s1,s2)
      equation
        s1cod = transStmt(s1);
        s2cod = transStmt(s2);
        l1 = genlabel();
        l2 = genlabel();
        compcod = transComparison(comp, l1);
        cod3 = listAppend6(
           compcod, s1cod,
           {Mcode.MJMP(l2)},
           {Mcode.MLABEL(l1)},
           s2cod,
           {Mcode.MLABEL(l2)} );
      then cod3;

    // while ...
    case Absyn.WHILE(comp,s1)
      equation
        bodycod = transStmt(s1);
        l1 = genlabel();
        l2 = genlabel();
        compcod = transComparison(comp, l2);
        cod3 = listAppend5(
          {Mcode.MLABEL(l1)},
          compcod, bodycod,
          {Mcode.MJMP(l1)},
          {Mcode.MLABEL(l2)} );
      then cod3;

    // to e1 do s1 ...
    case Absyn.TODO(e1,s1)
      equation
        tocod = transExpr(e1);
        bodycod = transStmt(s1);
```

```
        t1 = gentemp();
        l1 = genlabel();
        l2 = genlabel();
        cod3 = listAppend10(
          tocod,
          {Mcode.MSTO(t1)},
          {Mcode.MLABEL(l1)},
          {Mcode.MLOAD(t1)},
          {Mcode.MB(Mcode.MSUB(),Mcode.N(1))},
          {Mcode.MJ(Mcode.MJN,l2)},
          {Mcode.MSTO(t1)},
          bodycod,
          {Mcode.MJMP(l1)},
          {Mcode.MLABEL(l2)} );
      then cod3;

    // read {}
    case Absyn.READ({})  then {};

    // read id1,id2,...
    case Absyn.READ(id :: idListRest)
      equation
        cod2 = transStmt(Absyn.READ(idListRest));
      then Mcode.MGET(Mcode.I(id) :: cod2);

    // write {}
    case Absyn.WRITE({})then {};

   // write id1,id2,...
    case Absyn.WRITE(id :: idListRest)
      equation
        cod2 = transStmt(Absyn.WRITE(idListRest));
      then Mcode.MPUT(Mcode.I(id) :: cod2);

    // sequence of statements: stmt1 ; stmt2
    case Absyn.SEQ(stmt1,stmt2)
      equation
        cod1 = transStmt(stmt1);
        cod2 = transStmt(stmt2);
        cod3 = listAppend(cod1, cod2);
      then cod3;

  end matchcontinue;

end transStmt;
```

3.1.6.4 Emission of Textual Assembly Code

The translational semantics of PAM is specified as a translation from abstract syntax to a sequence of machine instructions in the structured MCode representation. However, we would like to emit the machine instructions in a textual assembly form. The conversion from the MCode representation to the textual assembly form is accomplished by the `emitAssembly` function and associated functions below. This is not really part of the translational semantics. Here, MetaModelica is used as a semi-functional programming language, to implement the desired conversion. The `print` primitive has been included in the standard MetaModelica library for such purposes.

```
package Emit
"Print out the MCode in textual assembly format
 Note: this is not really part of the specification of PAM semantics"

import Mcode;

function emitAssembly  "Print an MCode instruction"
  input List<Mcode.MCode> inMCodeList;
algorithm
```

```
  _ := matchcontinue (inMCodeList)
    local
      Mcode.MCode instr;
      MCodeList rest;

    case ({}) then ();

    case (instr :: rest)
      equation
        emitInstr(instr);
        emitAssembly(rest);
      then ();

  end matchcontinue;
end emitAssembly;

function emitInstr
  input Mcode.MCode inMCode;
algorithm
  _ := matchcontinue (in_MCode)
    local
      String op;
      Mcode.MBinOp mbinop;
      Mcode.MOperand mopr,mlab;
      Mcode.MCondJmp jmpop;

    // Print an MCode instruction
    case (Mcode.MB(mbinop,mopr))
      equation
        op = mbinopToStr(mbinop);
        emitOpOperand(op, mopr);
      then ();

    case (Mcode.MJ(jmpop,mlab))
      equation
        op = mjmpopToStr(jmpop);
        emitOpOperand(op, mlab);
      then ();

    case (Mcode.MJMP(mlab))
      equation
        emitOpOperand("J", mlab);
      then ();

    case (Mcode.MLOAD(mopr))
      equation
        emitOpOperand("LOAD", mopr);
      then ();

    case (Mcode.MSTO(mopr))
      equation
        emitOpOperand("STO", mopr);
      then ();

    case (Mcode.MGET(mopr))
      equation
        emitOpOperand("GET", mopr);
      then ();

    case (Mcode.MPUT(mopr))
      equation
        emitOpOperand("PUT", mopr);
      then ();

    case (Mcode.MLABEL(mlab))
      equation
        emitMoperand(mlab);
```

```
        print("\tLAB\n");
      then ();

    case (Mcode.MHALT())
      equation
        print("\tHALT\n");
      then ();

  end matchcontinue;
end emitInstr;

function emitOpOperand
  input String opstr;
  input Mcode.MOperand mopr;
algorithm
  print("\t");
  print(opstr);
  print("\t");
  emitMoperand(mopr);
  print("\n");
end emitOpOperand;

function emitInt
  input Integer i;
protected
  String s;
algorithm
  s := intString(i);
  print(s);
end emitInt;

function emitMoperand
  input Mcode.MOperand inMOperand;
algorithm
  _ := matchcontinue (inMOperand)
    local String id; Integer number,labno,tempnr;
    case (Mcode.I(id))     equation print(id); then ();
    case (Mcode.N(number)) equation emitInt(number); then ();
    case (Mcode.L(labno))  equation print("L"); emitInt(labno); then ();
    case (Mcode.T(tempnr)) equation print("T"); emitInt(tempnr); then ();
  end matchcontinue;
end emitMoperand;

function mbinopToStr
  input Mcode.MBinOp inMBinOp;
  output String outString;
algorithm
  outString := matchcontinue (inMBinOp)
    case (Mcode.MADD())  then "ADD";
    case (Mcode.MSUB())  then "SUB";
    case (Mcode.MMULT()) then "MULT";
    case (Mcode.MDIV())  then "DIV";
  end matchcontinue;
end mbinopToStr;


function mjmpopToStr
  input Mcode.MCondJmp inMCondJmp;
  output String outString;
algorithm
  outString := matchcontinue (inMCondJmp)
    case (Mcode.MJNP()) then "JNP";
    case (Mcode.MJP())  then "JP";
    case (Mcode.MJN())  then "JN";
    case (Mcode.MJNZ()) then "JNZ";
    case (Mcode.MJPZ()) then "JPZ";
    case (Mcode.MJZ())  then "JZ";
```

```
  end matchcontinue;
end mjmpopToStr;

end Emit;
```

3.1.6.5 Translate a PAM Program and Emit Assembly Code

The main function below performs the full process of translating a PAM program to textual assembly code, emitted on the standard output file. First, the PAM program is parsed, then translated to Mcode, which subsequently is converted to textual form.

```
package Main
  import Parse;
  import Trans;
  import Emit;

function main
  "Parse and translate a PAM program into Mcode,
   then emit it as textual assembly code."
protected
  type MCodeList = List<Mcode.MCode>;
  Absyn.Stmt program;
  MCodeList mcode;
algorithm
  program := Parse.parse();
  mcode := Trans.transProgram(program);
  Emit.emitAssembly(mcode);
end main;

end Main;
```

3.2 The Semantics of Mcode

In order to have a complete translational semantics of PAM, the meaning of each Mcode instruction must also be specified. This can be accomplished by an interpretive semantic definition of Mcode in MetaModelica.

However, we abstain from giving semantic definitions of machine code instruction sets for now since the current focus is the translation process, but may return to this topic later.

3.3 Translational Semantics for Symbolic Differentiation

Symbolic differentiation of expressions is a translational mapping that transforms expressions into differentiated expressions.

```
uniontype Exp
  record RCONST  Real x1;               end RCONST;
  record PLUS    Exp x1; Exp x2;        end PLUS;
  record SUB     Exp x1; Exp x2;        end SUB;
  record MUL     Exp x1; Exp x2;        end MUL;
  record DIV     Exp x1; Exp x2;        end DIV;
  record NEG     Exp x1;                end NEG;
  record IDENT   String name;           end IDENT;
  record CALL    Exp id; List<Exp> args; end CALL;
  record AND     Exp x1; Exp x2;        end AND;
  record OR      Exp x1; Exp x2;        end OR;
  record LESS    Exp x1; Exp x2;        end LESS;
  record GREATER Exp x1; Exp x2;        end GREATER;
end Exp;
```

An example function `difft` performs symbolic differentiation of the expression `expr` with respect to the variable `time`, returning a differentiated expression. In the patterns, _ underscore is a reserved word that can be used as a placeholder instead of a pattern variable when the particular value in that place is not needed later as a variable value. The **`as`**-construct: `id as IDENT(_)` in the third of-branch is used to bind the additional identifier `id` to the relevant expression.

We can recognize the following well-known derivative rules represented in the match-expression code:

- The time-derivative of a constant (`RCONST()`) is zero.
- The time-derivative of the `time` variable is one.
- The time-derivative of a time dependent variable `id` is `der(id)`, but is zero if the variable is not time dependent, i.e., not in the list `tvars`/`timevars`.
- The time-derivative of the sum (`add(e1,e2)`) of two expressions is the sum of the expression derivatives.
- The time-derivative of `sin(x)` is `cos(x)*x'` if `x` is a function of time, and `x'` its time derivative.
- etc...

We have excluded some operators in the `difft` example.

```
function difft "Symbolic differentiation of expression with respect to time"
  input  Exp expr;
  input  List<IDENT> timevars;
  output Exp diffexpr;
algorithm
  diffexpr := matchcontinue (expr, timevars)
    local
      Exp e1prim,e2prim,tvars;
      Exp e1,e2,id;
    // der of constant
    case(RCONST(_), _) then RCONST(0.0);
    // der of time variable
    case(IDENT("time"), _) then RCONST(1.0);
    // der of any variable id
    case (id as IDENT(_), tvars)
      then
      if listMember(id,tvars)
      then CALL(IDENT("der"),List(id))
      else RCONST(0.0);
    // (e1 + e2)' => e1' + e2'
    case (ADD(e1,e2), tvars)
      equation
        e1prim = difft(e1,tvars);
        e2prim = difft(e2,tvars);
      then ADD(e1prim,e2prim);
    // (e1 - e2)' => e1' - e2'
    case (SUB(e1,e2),tvars)
      equation
       e1prim = difft(e1,tvars);
       e2prim = difft(e2,tvars);
      then SUB(e1prim,e2prim);
    // (e1 * e2)' => e1' * e2 + e1 * e2'
    case (MUL(e1,e2),tvars)
      equation
       e1prim = difft(e1,tvars);
       e2prim = difft(e2,tvars);
      then PLUS(MUL(e1prim,e2),MUL(e1,e2prim));
    // (e1 / e2)' => (e1' * e2 – e1 * e2') / e2 * e2
    case (DIV(e1,e2),tvars)
      equation
        e1prim = difft(e1,tvars);
        e2prim = difft(e2,tvars);
      then DIV(SUB(MUL(e1prim,e2),MUL(e1,e2prim)), MUL(e2,e2));
    // (-e1)' => -e1'
```

```
    case (NEG(e1),tvars)
      equation
        e1prim = difft(e1,tvars);
      then NEG(e1prim);
    // sin(e1)' => cos(e1) * e1'
    case CALL(IDENT("sin"),List(e1)),tvars)
      equation
        e1prim = difft(e1,tvars);
      then MUL(CALL(IDENT("cos"),List(e1)),e1prim);
    // (e1 and e2)' => e1'and e2'
    case (AND(e1,e2),tvars)
      equation
        e1prim = difft(e1,tvars);
        e2prim = difft(e2,tvars);
      then AND(e1prim,e2prim);
    // (e1 or e2)' => e1' or e2'
    case (OR(e1,e2),tvars)
      equation
        e1prim = difft(e1,tvars);
        e2prim = difft(e2,tvars);
      then OR(e1prim,e2prim);
    // (e1<e2)' => e1' < e2'
    case (LESS(e1,e2),tvars)
      equation
        e1prim = difft(e1,tvars);
        e2prim = difft(e2,tvars);
      then LESS(e1prim,e2prim);
    // (e1 > e2)' => e1' > e2'
    case (GREATER(e1,e2),tvars)
      equation
        e1prim = difft(e1,tvars);
        e2prim = difft(e2,tvars);
      then GREATER(e1prim,e2prim);

    // etc...

  end matchcontinue;

end difft;
```

A related exercise can be found in Appendix D, with a corresponding solution in Appendix E.

3.4 Summary

This chapter introduced the concept of translational semantics, which was applied to the small PAM language. A translational semantics for translating PAM to a simple machine language was developed. The machine has only one register, and includes arithmetic instructions and conditional and unconditional jump instructions. A structured representation of the instruction set, called Mcode, was defined. Much of the translation is expressed through parameterized code templates within some of the MetaModelica match-expression cases. The complete PAMTRANS translational semantics is available in Appendix C, subsection 2.

The reader may have noted that we used many append instructions in the semantics, since the sequence of output code instructions is represented as a linked list. This can be avoided by an alternative way of representing the output code as an ordered sequence of instructions. For example, we can use a binary tree built by a binary sequencing operator (e.g. `MSEQ`), which can be obtained by for example adding an `MSEQ of Mcode * Mcode` operator declaration to the `Mcode` union type.

We have also shown a small set of translation rules for symbolic differentiation of mathematical expressions.

3.5 Exercises

See Appendix D.

Chapter 4

Getting Started – Practical Details

This chapter provides information about technical details that the reader will need to know in order to get started using MetaModelica for language specification and compiler/interpreter generation as well as other related tasks. This includes information about where the `OpenModelica` system resides, how to invoke the `omc` program generator, how to compile and link generated code, and (in the near future) how to run the Modelica debugger, etc.

We start in Section 4.3 with a set of very small executable examples which can be run interactively from OMNotebook – the OpenModelica electronic notebook, without any need for a lexer or parser generator.

In order to keep the presentation concise, we return to the simplest of all language examples described so far—the expression language Exp1 presented at the beginning of Chapter 2. We show how to build and run a working calculator that can evaluate constant arithmetic expressions expressed in the Exp1 language. We also describe how to build an interpreter for a larger language—the PAMDECL language described in Section 2.7.

4.1 Activating MetaModelica

The MetaModelica language features are activated by the following command to `omc` (The OpenModelica compiler):

```
setCommandLineOptions({"+g=MetaModelica"})
```

You can give this command within any of the OpenModelica subsystems including OMNotebook (see Section 4.3), OMEdit, the MDT Eclipse plugin, and OMShell.

4.2 Path and Locations of Needed Files

Many of the example models are available in the OpenModelica subversion repository which is referenced below. In order to use it, you need to provide a username (`anonymous`) and password (`none`, 4 letters).

4.2.1 Windows Dependencies

To run the examples in the notebook, only the OpenModelica system is required. It is available for download at `http://openmodelica.org/index.php/download/download-windows`.

The other examples using parsers need the OMDEV package. It is installed by fetching `https://openmodelica.org/svn/OpenModelica/installers/windows/OMDev` to `C:/OMDev`, and setting the environment variable `OMDEV` to `C:/OMDev`. Use the script in `C:\OMDev\tools\msys\msys.bat` to launch the shell that you use to run the examples later in this chapter.

4.2.2 Linux and Mac OSX Dependencies

Use the `openmodelica` package provided in the OpenModelica repositories for Debian/Ubuntu (http://openmodelica.org/index.php/download/download-linux) and Macports (http://openmodelica.org/index.php/download/download-mac). If your operating system of choice is not supported, build `openmodelica` from source. In order to run the examples using parsers, `flex` and `bison` also need to be installed. These are provided for most major operating systems.

4.2.3 Downloading the Examples

The reader may fetch the example files of this book from:
http://openmodelica.org/metamodelica/exercises/
More recent versions may be available at:
http://openmodelica.org/svn/OpenModelica/trunk/testsuite/meta/MetaModelicaDev
(user: anonymous, pass:none).

4.3 Examples of Using MetaModelica from OMNotebook

This example is available in the notebook `Basic-Exercise-MetaModelica.onb`, which covers examples of representing symbolic expressions as tree data structures declared as union types, and symbolic manipulation of such trees.

As mentioned previously, in order to use the examples, you need to instruct the OpenModelica Compiler omc to use the MetaModelica language extensions by giving the following command in a Modelica input cell in a notebook. Click in the cell and type the key shift-enter to evaluate the command.

```
setCommandLineOptions({"+g=MetaModelica"})
```

You can locate and use the abovementioned notebook, or create your own example notebook using the definitions described here.

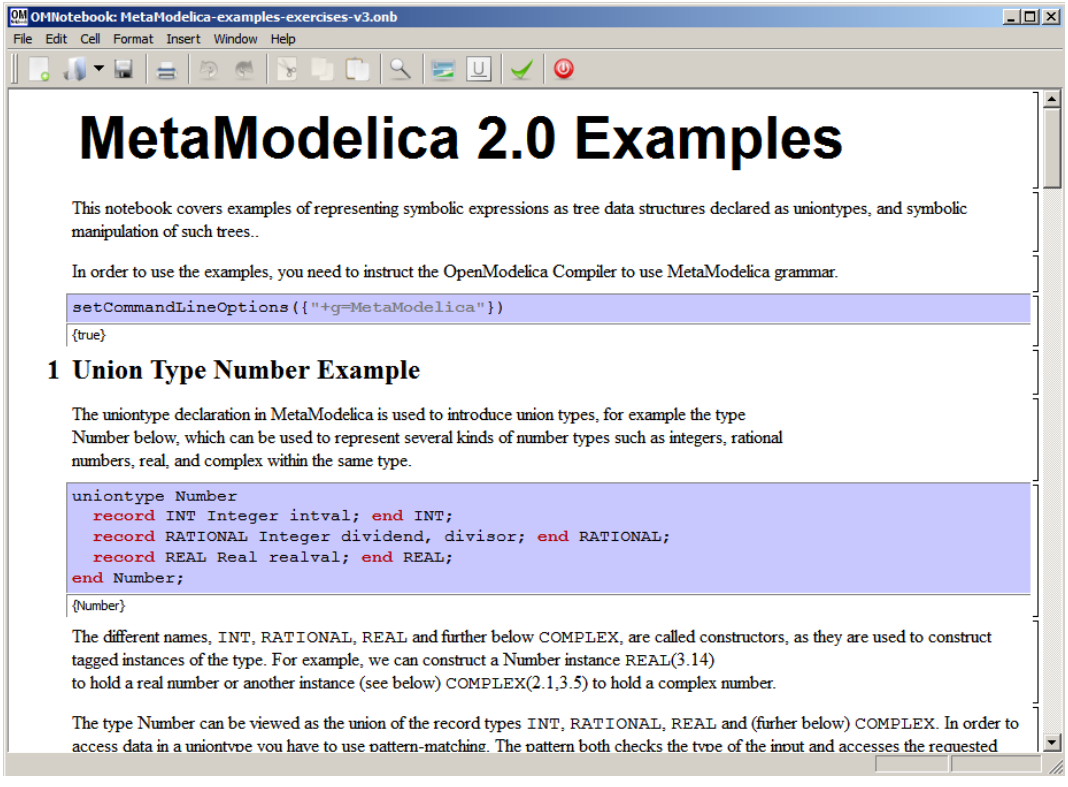

Figure 4-1. Example use of OMNotebook for MetaModelica examples and exercises.

4.3.1 Union Type Number Example

The `uniontype` declaration in MetaModelica is used to introduce union types, for example the type `Number` below, which can be used to represent several kinds of number types such as integers, rational numbers, real, and complex within the same type.

We type in the uniontype definition in a cell in the notebook, and evaluate it by typing shift-enter:

```
uniontype Number
  record INT Integer intval;                  end INT;
  record RATIONAL Integer dividend, divisor; end RATIONAL;
  record REAL Real realval;                   end REAL;
end Number;
```

The different names, `INT`, `RATIONAL`, `REAL` and further below `COMPLEX`, are called constructors, as they are used to construct tagged instances of the type. For example, we can construct a `Number` instance `REAL(3.14)` to hold a real number or another instance (see below) `COMPLEX(2.1,3.5)` to hold a complex number.

The type `Number` can be viewed as the union of the record types `INT`, `RATIONAL`, `REAL` and (further below) `COMPLEX`. In order to access data in a uniontype you have to use pattern-matching. The pattern both checks the type of the input and accesses the requested data.

We enter and evaluate the definition of the function `numberAbs` in a new cell in the notebook:

```
function numberAbs  "Returns the absolute value of a Number as a Real."
  input Number number;
  output Real result;
algorithm
  result := match number
    local
      Integer intval,dividend,divisor;
      Real realval,re,im;
    case INT(intval) then abs(intval);
    case RATIONAL(dividend,divisor) then abs(dividend/divisor);
    case REAL(realval) then abs(realval);
  end match;
end numberAbs;
```

We call the function:

```
numberAbs(INT(-13))
```

with result:

```
13.0
```

A few calls with corresponding results:

```
numberAbs(REAL(-1.3))
1.3
```

```
numberAbs(RATIONAL(13,55))
0.236363636363636
```

The most frequent use of union types in MetaModelica is to specify abstract syntax tree representations used in language specifications, e.g., as shown in Chapter 2 and in Section 4.4 below. Since union types are often used to represent tree structures, they may be recursive.

4.3.1.1 Small Number Exercise

In this exercise you should add a case for complex numbers. You can use the expression `abs(sqrt(im^2+re^2))` for computing the absolute value of the complex number, and the pattern `COMPLEX(im,re)` for matching the parts of the complex number to the pattern variables `im` and `re`. Update and evaluate the `Number` definition e.g. as follows:

```
uniontype Number
  record INT Integer intval;                  end INT;
  record RATIONAL Integer dividend, divisor; end RATIONAL;
```

```
  record REAL Real realval;                  end REAL;
  record COMPLEX Real re,im;                 end COMPLEX;
end Number;
```

The function numberAbs below is incomplete. You should add a case for COMPLEX numbers to compute the absolute value as just described.

```
function numberAbs  "Returns the absolute value of a Number as a Real."
  input Number number;
  output Real result;
algorithm
  result := match number
    local
      Integer intval,dividend,divisor;
      Real realval,re,im;
    case INT(intval) then abs(intval);
    case RATIONAL(dividend,divisor) then abs(dividend/divisor);
    case REAL(realval) then abs(realval);
    case COMPLEX //... fill in
  end match;
end numberAbs;
```

You should be able to call the function numberAbs for the example as follows:

```
numberAbs(COMPLEX(1.3,-4.2))
```

giving the result

```
4.39658958739612
```

4.3.2 Exp1Real Calculator in the Notebook

The Exp1 calculator example from Section 2.1 and Section 4.5.3 can be defined and used directly in the notebook if we avoid building a lexer and parser. However, we change the uniontype from Exp to ExpReal by adding an RCONST node to be able to represent Real constants and change the result of the function to have Real type. First enter the needed uniontype definition:

```
uniontype ExpReal
  record  INTconst Integer int;        end INTconst;
  record  Rconst   Real real           end Rconst;
  record  ADDop    Exp exp1;  Exp exp2; end ADDop;
  record  SUBop    Exp exp1;  Exp exp2; end SUBop;
  record  MULop    Exp exp1;  Exp exp2; end MULop;
  record  DIVop    Exp exp1;  Exp exp2; end DIVop;
  record  NEGop    Exp exp;            end NEGop;
end ExpReal;
```

Then enter the function definition:

```
function eval
  input  ExpReal  inExp;
  output Real     res;
algorithm
  res := match inExp
    local Integer v1; Real r; ExpReal e1,e2;
    case INTconst(v1) then v1;
    case RCONST(r)    then r;
    case ADDop(e1,e2) then eval(e1) + eval(e2);
    case SUBop(e1,e2) then eval(e1) - eval(e2);
    case MULop(e1,e2) then eval(e1) * eval(e2);
    case DIVop(e1,e2) then eval(e1) / eval(e2);
    case NEGop(e1)    then -eval(e1);
  end match;
end eval;
```

Call the function on a small expression tree:

```
// 42.0 * (1.5 + (2 - 1.9))
eval(MULop(RCONST(42), ADDop(RCONST(1.5), SUBop(RCONST(2), RCONST(1.9)))))
```

giving the result:

```
67.2
```

4.3.3 List Data Structures

Lists are common data structures in declarative languages since they conveniently allow representation and manipulation of sequences of elements. Elements can be efficiently (in constant time) added to beginning of lists in a declarative way. All elements of a list have the same type. The following basic list construction operators are available, presented below through examples. See also Section 6.2 for more information about lists.

In the examples, on the first line the expression to be evaluated is shown, on the following line the evaluated result.

```
{} // The empty list
{}
```

```
{1, 2, 3} // The list constructor
{1, 2, 3}
```

```
1 :: {2, 3} // The cons :: operator
{1, 2, 3}
```

```
1 :: "2" :: {} // An illegal expression
```

```
OMC-ERROR:
"Error: Failed to match types of cons expression 1::"2"::{}. The head has type
Integer and the tail list<String>.
"
```

Appending lists is expensive; it uses the cons operation once for each element in the first argument

```
listAppend( {1, 2, 3}, {4})
{1, 2, 3, 4}
```

The types of lists often need to be specified. Named list types can be declared using MetaModelica type declarations.

```
type StringList = list<String>;
{StringList}
```

The most readable and convenient way of accessing elements in an existing list or constructing new lists is through pattern matching operations, used in the `stringDelimitList` function below.

```
function stringDelimitList
  input StringList stringList;
  input String delimiter;
  output String outString;
algorithm
  outString := match stringList
    local
      String head;
      StringList tail;
    // Remember to match the empty list
    case {} then "" ;
    // The last element does not need the delimiter
    case {head} then head;
    // Pattern-matching using the cons operator
    case head::tail then head + delimiter + stringDelimitList(tail,delimiter);
  end match;
end stringDelimitList;
```

Calling the function:

```
stringDelimitList( {"1", "2", "3", "4"}, "," )
```

with result

```
"1,2,3,4"
```

4.3.4 Tuples

Tuples are represented by parenthesized, comma-separated sequences of items each of which may have a different type. For example:

```
(  1, 2.0, "3" )
( 12, 2.6, "mine" )
```

Named tuple types can be declared explicitly through the type declaration using the tuple type constructor.

```
type ThreeTup = tuple<Integer,Real,String>;
{ThreeTup}
```

The values of a tuple are easiest accessed through pattern-matching.

```
function threeTupString
  input ThreeTup tup;
  output String str;
algorithm
  str := match tup
    local Integer int; Real real; String string;
    case (int,real,string)
      then stringAppendList({"(", String(int), ", ",
                             String(real), ", ", string, ")"});
  end match;
end threeTupString;
```

Calling the function:

```
threeTupString( ( 1, 2.5, "3" ) )
```

```
"(1, 2.5, 3)"
```

4.3.5 Option Types

Option types have been introduced in MetaModelica to provide a type-safe way of representing the common situation where a data item is optionally present in a data structure.

The constructor `NONE()` is used to represent the case where the optional data item is not present, whereas the constructor `SOME()` is used when the data item is present in the data structure.

We start by defining an `Option` type and a function using it:

```
type StringOption = Option<String>;
```

```
function stringOrDefault
  input StringOption strOpt;
  input String default;
  output String str;
algorithm
  str := match strOpt
    case SOME(str) then str;
    else default;
  end match;
end stringOrDefault;
```

Calling the function a few times:

```
stringOrDefault(NONE(),"default")
"default"
```

```
stringOrDefault(SOME("string"),"default")
```

```
"string"
```

4.4 The MDT Eclipse Plugin

To run the following exercises, we recommend using the MDT Eclipse plug-in. For information about using and installing MDT see the OpenModelica User's Guide (Fritzson et al 2011), especially if you would like to use the debugger for stepping through the examples interactively.

The Modelica Development Tooling (MDT) Eclipse plug-in as part of OMDev – The OpenModelica Development Environment which integrates the OpenModelica compiler with Eclipse. MDT, together with the OpenModelica compiler, provides an environment for working with Modelica and MetaModelica development projects. The following features are available:

- Browsing support for Modelica projects, packages, and classes
- Wizards for creating Modelica projects, packages, and classes
- Syntax color highlighting
- Syntax checking
- Browsing of the Modelica Standard Library or other libraries
- Code completion for class names and function argument lists.
- Goto definition for classes, types, and functions.
- Displaying type information when hovering the mouse over an identifier.

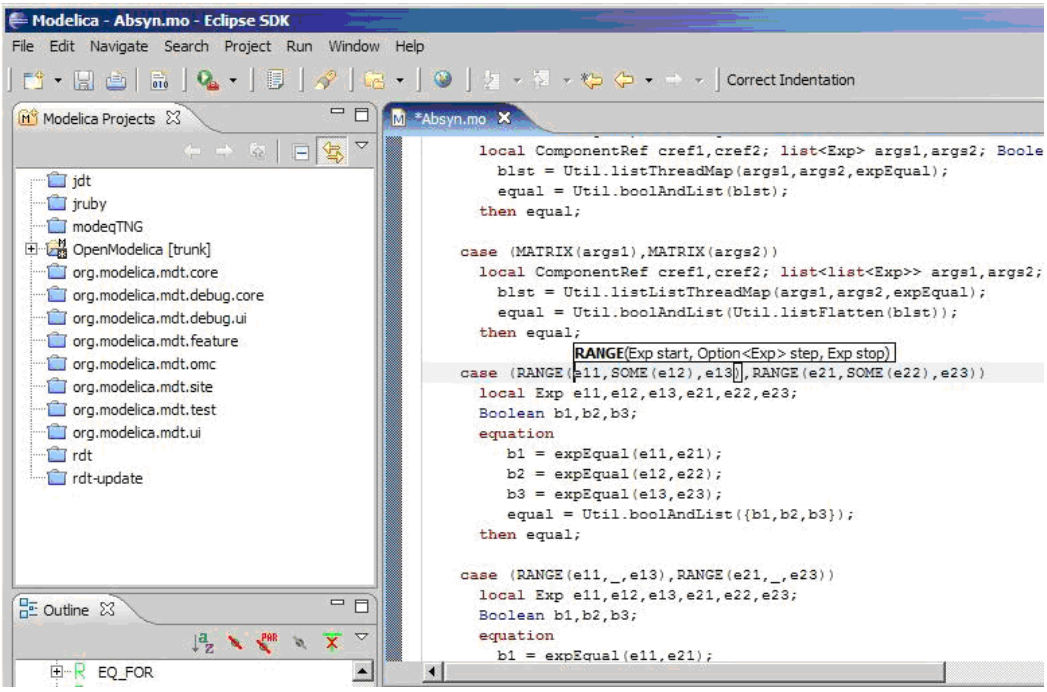

Figure 5-2. MDT usage example. Code assistance when writing cases with records in MetaModelica.

4.5 The Exp1 Calculator Again

4.5.1 Running the Exp1 Calculator

Before building the Exp1 calculator it is instructive to show how it can be used. The calculator is compiled and executed from within OpenModelica simply by calling the main function from a script.

Typing `make –f Makefile.omc run` will run the calculator using the contents of the file `program.txt` as input.

```
>> make –f Makefile.omc run
***   program   ***
1 + 6 / 3 + 5
*** end program ***
...
8
```

Result: 1 + 6 / 3 + 5 = 8

The following example shows how the calculator reacts when it is fed an expression which does not belong to the Exp1 expression language. Remember that this language only allows simple arithmetic expressions not including variables or symbolic constants.

```
>> make –f Makefile.omc run
***   program   ***
1 + 6 / 3 + 5 + hej
*** end program ***

...
Syntax error at or near line 1.
```

4.5.2 Building the Exp1 Calculator

Before building the Exp1 calculator, we need to locate the OpenModelica, Flex and Bison tools. It is useful for the reader who wishes to test building and running the calculator to create his/her own work directory (e.g. called `myexp1`).

4.5.2.1 Source Files to be Provided

Three files are needed to specify all properties (syntax and semantics) of the Exp1 language. One additional file defines the main program.

- The file `Exp1.mo` contains an interpretive style MetaModelica specification and abstract syntax of the Exp1 language in MetaModelica form, here within the single MetaModelica package Exp1.
- The file parser.y contains the grammar of the Exp1 language in Yacc-style BNF form.
- The file lexer.l specifies the lexical syntax of tokens in the Exp1 language in Lex-style regular expression form.
- In addition, `main.mo` and `Parse.mo` define the C main program that calls initialization routines, the generated scanner, parser and evaluator, and prints the evaluated result.

4.5.2.2 Generated Source Files

The following five files are generated by the MetaModelica system and the Yacc and Lex tools, respectively:

- The files `Exp1_eval.*` are generated by the `omc` translator using the `omc` command.
- The files `parser.c` and `parser.h` are generated by Bison, and contain a parser for Exp1 and token definitions, respectively.
- The file `lexer.c` is generated by Flex, and contains a scanner for Exp1.

4.5.2.3 Library File(s)

The following system specific library files and header files are used.

- The file `meta_modelica.h` contains definitions and macros for calling the MetaModelica runtime system and predefined functions (located in `$OPENMODELICAHOME/include/omc`).
- The file `libc_runtime.a` is a library of all MetaModelica runtime system routines and predefined functions (located in `$OPENMODELICAHOME/lib/omc`).

4.5.2.4 Makefile for Building the Exp1 Calculator

Building the Exp1 calculator from the needed components is conveniently described by a `Makefile`, such as the one below. The GNU Compiler Collection is used here. Library files and header files are found in `$OPENMODELICAHOME/{include,lib}`. The usual make dependencies are specified. The command:

```
make –f Makefile.omc run
```

will build and run the calculator (called `executable`) whereas the command:

```
make –f Makefile.omc clean
```

will remove all generated files, object files and the binary executable file.

```
# Makefile.omc
# Makefile for running the Exp1 calculator
CLEAN=Exp1_* Parse_*
SOLUTIONS=SCRIPT.mos
DEPS=lexer.o parser.o
include ../common.omc
ifdef OMDEV
FPIC=
else
FPIC=-fPIC
endif

ifndef OPENMODELICAHOME
CFLAGS = -I'../../../../build/include/omc' $(FPIC)
else
CFLAGS = -I'$(OPENMODELICAHOME)/include/omc' $(FPIC)
endif

CC = gcc
run: run2 clean

OMC = ${OPENMODELICAHOME}/bin/omc +d=rml,noevalfunc +g=MetaModelica
TEST = $(shell if [ -f ../../../rtest ]; then echo ../../../rtest; else echo $(O
MC); fi) +d=rml,noevalfunc +g=MetaModelica

# LEXER

lexer.o:  lexer.c parser.h
lexer.c:  lexer.l
        flex -t -l lexer.l >lexer.c.tmp
        @mv lexer.c.tmp lexer.c

# PARSER

parser.o:  parser.c parser.h
parser.c parser.h:  parser.y
        bison -d parser.y
        @mv parser.tab.c parser.c
        @mv parser.tab.h parser.h

clean:
        @rm -f $(CLEAN) *.o *.so *.dll *.exe parser.c parser.h lexer.c rtest *~

run2: $(DEPS)
        @echo "***   program   ***"
        @cat program.txt
```

```
	@echo "*** end program ***"
	$(OMC) SCRIPT.mos < program.txt
```

4.5.3 Source Files for the Exp1 Calculator

Below we present the three source files `lexer.l`, `parser.y`, and `Exp1.mo`, needed to specify the syntax and semantics of the Exp1 language, as well as the main program files `SCRIPT.mos` and `Parse.mo`.

4.5.3.1 Lexical Syntax: lexer.l

The file `lexer.l` defines the lexical syntax of the Exp1 language, identical to what was presented in Section 2.1.1, but augmented by mentioning necessary include files.

The global variable `yylval` is used to transmit the values of tokens that have values—such as integer constants (`T_INTCONST`)—to the parser.

Character sequences including new line (`\n`) which cannot give rise to legal tokens in Exp1 are taken care of by junk, which is just skipped.

The routine `Exp1__INTconst` builds abstract syntax integer leaf nodes and is generated by the OpenModelica compiler when processing the abstract syntax definitions in `Exp1.mo`.

The routine `mmc_mk_icon` (from `meta_modelica.h`) builds MetaModelica compatible integer constants that can be passed to MetaModelica constructors such as `Exp1.INTconst`, here callable as `Exp1__INTconst`.

```
/* file lexer.l */
%{
#define YYSTYPE void*
#include "parser.h"

void* absyn_integer(char *s);

#ifdef RML
#include "yacclib.h"
#include "Exp1.h"
#else
#include "meta_modelica.h"
extern struct record_description Exp1_Exp_INTconst__desc;
#define Exp1__INTconst(X1) (mmc_mk_box2(3,&Exp1_Exp_INTconst__desc,(X1)))
#endif

%}

digit           ("0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9")
digits          {digit}+
junk            \n
letter          [$_A-Za-z.]
letters         {letter}+


%%

{digits}     { yylval = absyn_integer(yytext); return T_INTCONST; }
"+"          return T_ADD;
"-"          return T_SUB;
"*"          return T_MUL;
"/"          return T_DIV;
"^"          return T_POW;
"!"          return T_FACTORIAL;
"("          return T_LPAREN;
")"          return T_RPAREN;
{junk}+         ;
{letters}    return T_ERR;
```

```
%%

void* absyn_integer(char *s)
{
  return Exp1__INTconst(mmc_mk_icon(atoi(s)));
}
```

4.5.3.2 Grammar: parser.y

The grammar file `parser.y` follows below. The grammar rules are identical to those presented in Section 2.1.1. However, some include files are mentioned here and tree-building calls have been inserted at the parser rules in order to build the abstract syntax tree during parsing.

The tree building routines `Exp1__ADDop`, `Exp1__SUBop`, `Exp1__MULop`, `Exp1__DIVop`, `Exp1__NEGop`, and `Exp1__INTconst` match the runtime definitions of the Exp1 abstract syntax in the package `Exp1` that can be found in the file `Exp1.mo`. Leaf nodes such as `INTconst` are returned by the scanner.

```
/* file  parser.y */
%{
#include <stdio.h>
#include <stdlib.h>

#define YYSTYPE void*
void* absyntree;

int yyerror(char *s)
{
  extern int yylineno;
  fprintf(stderr,"Syntax error at or near line %d.\n",yylineno);
  exit(1);
}

int yywrap()
{
  return 1;
}

#ifdef RML
#include "Exp1.h"
#include "yacclib.h"
#ifndef Exp1__FACop
#define Exp1__FACop(X) (void*)yyerror("")
#endif
#ifndef Exp1__POWop
#define Exp1__POWop(X,Y) (void*)yyerror("")
#endif
#else
#include "meta_modelica.h"
void* getAST()
{
  return absyntree;
}

extern struct record_description Exp1_Exp_ADDop__desc;
extern struct record_description Exp1_Exp_SUBop__desc;
extern struct record_description Exp1_Exp_MULop__desc;
extern struct record_description Exp1_Exp_DIVop__desc;
extern struct record_description Exp1_Exp_NEGop__desc;

const char* WORKAROUND__Exp1_Exp_POWop__desc__fields[] = {"exp1","exp2"};
struct record_description WORKAROUND__Exp1_Exp_POWop__desc = {
    "Exp1_Exp_POWop",
    "Exp1.Exp.POWop",
    WORKAROUND__Exp1_Exp_POWop__desc__fields
};
```

```
const char* WORKAROUND__Exp1_Exp_FACop__desc__fields[] = {"exp"};
struct record_description WORKAROUND__Exp1_Exp_FACop__desc = {
    "Exp1_Exp_FACop",
    "Exp1.Exp.FACop",
    WORKAROUND__Exp1_Exp_FACop__desc__fields
};

#define Exp1__ADDop(X1,X2) (mmc_mk_box3(4,&Exp1_Exp_ADDop__desc,(X1),(X2)))
#define Exp1__SUBop(X1,X2) (mmc_mk_box3(5,&Exp1_Exp_SUBop__desc,(X1),(X2)))
#define Exp1__MULop(X1,X2) (mmc_mk_box3(6,&Exp1_Exp_MULop__desc,(X1),(X2)))
#define Exp1__DIVop(X1,X2) (mmc_mk_box3(7,&Exp1_Exp_DIVop__desc,(X1),(X2)))
#define Exp1__NEGop(X1)    (mmc_mk_box2(8,&Exp1_Exp_NEGop__desc,(X1)))
#define Exp1__POWop(X1,X2)
(mmc_mk_box3(9,&WORKAROUND__Exp1_Exp_POWop__desc,(X1),(X2)))
#define Exp1__FACop(X1)
(mmc_mk_box2(10,&WORKAROUND__Exp1_Exp_FACop__desc,(X1)))
#endif

%}

%token T_INTCONST
%token T_LPAREN T_RPAREN
%token T_ADD
%token T_SUB
%token T_MUL
%token T_DIV
%token T_GARBAGE
%token T_ERR

%token T_POW
%token T_FACTORIAL

%%

/* Yacc BNF Syntax of the expression language Exp1 */
program
                 :  expression
                    { absyntree = $1; }

expression       :  term
                 |  expression  T_ADD  term
                    { $$ = Exp1__ADDop($1,$3);}
                 |  expression  T_SUB  term
                    { $$ = Exp1__SUBop($1,$3);}

term             :  u_element
                 |  term  T_MUL  u_element
                    { $$ = Exp1__MULop($1,$3);}
                 |  term  T_DIV  u_element
                    { $$ = Exp1__DIVop($1,$3);}

u_element        :  element
                 |  T_SUB  element
                    { $$ = Exp1__NEGop($2);}
                 |  T_FACTORIAL  element
                    { $$ = Exp1__FACop($2);}
                 |  element T_POW  u_element
                    { $$ = Exp1__POWop($1,$3);}

element          :  T_INTCONST
                 |  T_LPAREN  expression  T_RPAREN
                    { $$ = $2;}
```

4.5.3.3 Semantics: Exp1.mo

The abstract syntax and semantics of the small expression language Exp1 appears below, identical to the definitions in Section 2.1.2 and Section 2.1.4. Both have been placed in the MetaModelica package `Exp1`. For larger specifications it is customary to place the definition of abstract syntax in a package of its own. Note that the abstract syntax specification has been placed in the interface sections since the constructors need to be exported to be callable by the parser.

```
// file Exp1.mo
package Exp1 "Abstract syntax of the language Exp1 as defined using Modelica"

uniontype Exp
  record  INTconst Integer int;        end INTconst;
  record  ADDop    Exp exp1;  Exp exp2; end ADDop;
  record  SUBop    Exp exp1;  Exp exp2; end SUBop;
  record  MULop    Exp exp1;  Exp exp2; end MULop;
  record  DIVop    Exp exp1;  Exp exp2; end DIVop;
  record  NEGop    Exp exp;            end NEGop;
end Exp;

// Evaluation semantics of Exp1
function eval
  input  Exp     inExp;
  output Integer outInteger;
algorithm
 outInteger := match inExp
   local
     Integer v1,v2;
     Exp     e1,e2;

   case INTconst(v1) then v1;

   case ADDop(e1,e2)
     equation
       v1 = eval(e1); v2 = eval(e2);
     then v1 + v2;

   case SUBop(e1,e2)
     equation
       v1 = eval(e1); v2 = eval(e2);
     then v1 - v2;

   case MULop(e1,e2)
     equation
       v1 = eval(e1); v2 = eval(e2);
     then v1 * v2;

   case DIVop(e1,e2)
     equation
       v1 = eval(e1); v2 = eval(e2);
     then v1 / v2;

   case NEGop(e1)
     equation
       v1 = eval(e1);
     then –v1;

   end match;

end eval;
```

4.5.3.4 Calling the calculator

`SCRIPT.mos` is an OpenModelica script-file that is used to call the calculator. The Parse package is used to read input from the terminal and creates an abstract syntax tree using external C functions. This tree is then passed to the calculator.

```
/* SCRIPT.mos */
loadFile("Exp1.mo");
loadFile("Parse.mo");
getErrorString();
print("[Parse. Enter an expression, then press CTRL+z (Windows) or CTRL+d
(Linux).]
");
if true then
  syntree := Parse.parse();
end if;
// syntree; // Uncomment if you want to debug the AST
getErrorString();
i := Exp1.eval(syntree);
getErrorString();

/* Parse.mo */
package Parse

import Exp1;

protected
function yyparse
  output Integer i;
  external "C" annotation(Library = {"lexer.o","parser.o"});
end yyparse;

function getAST
  output Exp1.Exp exp;
  external "C" annotation(Library = {"lexer.o","parser.o"});
end getAST;

public
function parse
  output Exp1.Exp exp;
algorithm
  0   := yyparse();
  exp := getAST();
end parse;

end Parse;
```

4.5.4 Library Files

4.5.4.1 meta_modelica.h

The header file `meta_modelica.h` declares a number of primitive routines which are primarily used in the course of building abstract syntax trees during parsing.

The routines `mmc_mk_icon`, `mmc_mk_rcon`, `mmc_mk_scon` create MetaModelica representations for integers, real numbers and strings, respectively.

List construction is provided by `mmc_mk_cons` which creates a list cell and `mmc_mk_nil` which creates a nil pointer to represent the end of a list. The `mmc_mk_none` and `mmc_mk_some` constructors are used for the builtin MetaModelica `Option` type which is convenient for representing optional syntactic constructs.

Finally, the routines `mmc_mk_box0` to `mmc_mk_box9` construct abstract syntax tree nodes of arity 0 to 9. These should not be called directly, however. Instead use `generateHeader()` API call in OpenModelica to create macros.

4.6 An Evaluator for PAMDECL

4.6.1 Running the PAMDECL Evaluator

The executable is named `executable`, and is invoked by typing `executable` at the Linux prompt (>>). Input typed by the user is shown in boldface.

```
>> cat | executable

program
  a:   integer;
  foo: real;
body
  a   := 17;
  foo := a * 2 + 8;
  write foo;
end program
^D 42.0
```

Supplied with PAMDECL are a number of test programs located in subdirectory `prg/`. To run `prg5` type the following:

```
>> executable > prg/prg5

1.01
1.0201
1.04060401
1.08285670562808
1.1725786449237
1.3749406785311
1.89046186947955
3.57384607995613
12.7723758032178
163.133583658624
26612.5661173053
708228675.347948
```

4.6.2 Building the PAMDECL Evaluator

The following files are needed for building PAMDECL: `Absyn.mo` , `Env.mo`, `Eval.mo`, `lexer.l`, `parser.y`, `Main.mo`, `ScanParse.mo` and `Makefile.omc`.

The file contents are listed in Appendix C and can also be obtained by following the instructions in Section 4.2.3.

The executable is run by typing:

```
>> make –f Makefile.omc run
```

4.6.3 Calling C from MetaModelica

See the section on External Functions in the Modelica Language Specification and the examples of calling external C functions in the OpenModelica Users Guide. The type mapping for `List`, `Tuple`, `Option` and `uniontype` is `void*`. The macros used to construct and access these values are part of `meta_modelica.h`.

Chapter 5

Introductory Overview of MetaModelica 2.0

This chapter gives an introductory overview of the MetaModelica language elements that are not found in standard Modelica 3.2, as well as a short comparison between MetaModelica 1.0 and 2.0.

- Miscellaneous smaller features are introduced in Section 5.1, including the MetaModelica keywords, some predefined types, operators, and functions; handling arithmetic exceptions; possible constructs to be deprecated.
- Collection data structures such as lists, arrays, and sequences, and operations on those, are introduced in Chapter 6.
- Union types of recursive data structures used to build tree structures such as abstract syntax trees are introduced in Chapter 7, as well as tuples and Option types.
- The handling of failures and exceptions is presented in Chapter 8.
- The use of type inference is described in Chapter 9.
- Functions and various issues related to their use is presented in Chapter 10.
- Some enhancements regarding import from packages is described in Chapter 11.
- The OpenModelica text template language is presented in Chapter 12. This is a domain specific language strongly related to MetaModelica, and also compiling into MetaModelica.

5.1 Miscellaneous Features

5.1.1 MetaModelica Keywords

The following special MetaModelica *keywords* in addition to the ordinary Modelica 3 keywords are reserved words and may not be used as identifiers.

`_`	`as`	`case`	`guard`	`local`
`match`	`matchcontinue`	`uniontype`		

5.1.2 Predefined Types and Type Constructors

MetaModelica 2.0 includes the Modelica 3.1 predefined types, as well as the following types and type constructors. All type parameter values such as `Real` or `String` within <…> can be replaced by type variables.

`List<eltype>`	List of element type. Example: `List<Real>`
`Option<eltype>`	Option type of an element type. Example: `Option<String>`
`Tuple<t1,t2,…>`	Tuple type of one or more types. Example: `Tuple<Real, Integer, String>`

`Sequence`	An abstract sequence type introduced in Section 6.2.
`Any`	Type at the top of the type hierarchy. All other types are subtypes of Any.

5.1.3 Builtin MetaModelica Functions for Basic Boxed Data Types

New built-in functions are needed to perform operations on the MetaModelica boxed values. Boxed values are common in languages supporting symbolic processing. This means that

The functions are can be used directly in the MetaModelica code or they are generated in the C code from operators. We can classify these functions based on the types they operate on.

- Booleans: `boolEq`, `boolString`
- Integers: `intEq`, `intLt`, `intLte`, `intGt`, `intGte`, `intNe`, `intString`, `intAdd`, `intSub`, `intMul`, `intDiv`, `intMod`, `intMin`, `intMax`
- Reals: `realEq`, `realLt`, `realLte`, `realGt`, `realGte`, `realNe`, `realString`, `realAdd`, `realSub`, `realMul`, `realDiv`, `realMod`, `realMin`, `realMax`
- Strings: `stringEq`, `stringCompare`, `stringHash`, `stringGet`, `stringAppend`
- Lists: `listAppend`, `listMember`, `listGet`, `listReverse`, `cons` (:: operator), `stringAppendList`
- Standard Modelica arrays. Constructor: `array`(…) or {…}.
- MetaModelica globally mutable arrays: Constructor `MRArray`(…); Also: `mrarrayCreate`, `mrarrayGet`, `[ind]` square bracket index operator, `mrarrayCopy`, `mrarrayUpdate.` These are globally mutable dynamically allocated arrays. See also Section 5.3.

For a complete set of builtin operators and functions, including their signatures, see Appendix B.

5.1.5 Built-in Special Operators and Functions

MetaModelica introduces two new operators with special syntax:

Table 5-1. Special operators.

Operator Group	*Operator Syntax*	*Examples*
list element concatenation	`"a"::{"b","c"}` => `{"a","b","c"}`	`"a"::{"b","c"}` => `{"a","b","c"}`
as-operator	*ident* **`as`** pattern-*expr*	`x` **`as`** `INT(a, "name")`

Equality `=` and assignment `:=` are not expression operators since they are allowed only in equations and in assignment statements respectively. All binary expression operators are left associative.

The relational operators <, <=, >, >=, ==, <>, are only defined for scalar operands of simple types, not for arrays or records, as in standard Modelica.

There is also a generic structural value equality operator, `valueEq(expr1,expr2)`, (Table 5-3), giving `false` or `true`, which can be applied to values of primitive data types as well as to values of structured types such as arrays, lists, and trees. Also note the `referenceEq(expr1,expr2)` operator that compares the memory addresses (references) of the two expression value objects.

Table 5-2. Special equality operators.

`valueEq(…)`	Value equality, `valueEq`(*expr1*,*expr2*). The values of the two expressions are the same. The operator applies to both simple tupes and structured types such as array, lists, records, trees.
`referenceEq(…)`	Reference equality, `referenceEq`(*expr1*,*expr2*). The memory addresses (references) of the two expression value objects are the same. This operation is very efficient, but two different strings or nodes might have different references but

	the same values, e.g. `"a"` and `"a"+""` will typically have different references since they may have been allocated separately. Allocation addresses might also change during run-time if the data is moved by the run-time system.

The following built-in special operators in Modelica have the same syntax as a function call. However, they do *not* behave as mathematical functions since the result depends not only on the input arguments but also on the status of the simulation.

Table 5-3. Special operators with function syntax.

`failure(…)`	If the argument fails this function succeeds.
`fail(…)`	Generates a failure.

5.1.6 Arithmetic Exceptions

The effects of arithmetic overflow, underflow, or division by zero in Modelica are implementation dependent, depending on the C compiler and the Modelica tool in use. Either some value is produced and execution continues, or some kind of trap or exception is generated which can terminate execution if it is not handled by the application or the Modelica run-time system, e.g. catching failure. In MetaModelica several of these cases generate failures which can be handled within a match-expression, see Chapter 8.

5.1.7 MetaModelica Constructs to Possibly be Deprecated

The MetaModelica 2.0 language contains some constructs which may eventually be removed from the MetaModelica language. They are needed before better alternatives have been designed and implemented. The constructs that might eventually be depreciated are the following:

- Match-expressions with `matchcontinue` may eventually be replaced by match-expressions and exception handling and/or match-expressions with guards. The `matchcontinue` construct is more general, but these replacements often lead to more readable code.
- The fail and failure handling may be replaced by a full exception handling mechanism.
- The implicit import (Section 11.2.1) of uniontype record constructors into the scope of the uniontype declaration is planned to be removed. An explicit import statement (Section 11.2) will be needed to achieve the same effect.

5.2 Comparison Between MetaModelica 1.0 and 2.0

MetaModelica 2.0 contains the following enhancements and changes compared to MetaModelica 1.0:

- The pattern __ for constructors with arbitrary arguments, see Section 7.2.4 and Section 7.2.5.
- Guarded patterns in match-expressions, see Section 7.2.4.1.
- List and array comprehensions with patterns and guards, see Section 6.6.
- Generalization of for-loops to patterns with guards , see Section 6.6.1.
- Parameterized union types, see Section 7.1.2.
- Short < > syntax for type parameters in union types and functions, see Section 7.1.2.1 , Section 10.6.1, and below.
- Explicit import and access to union type record constructors without implicit import, see Section 11.2.
- Short explicit import of multiple items from packages, see Section 11.1.

- Local algorithm sections, see Section 7.2.1.1
- Type inference for as-bound declared variables, see Section 9.3.
- Error reporting location information, see Section 8.3.
- Functions with impure/pure keywords, see Section 10.3.
- Compact argument matching and named field access via dot notation, see Section 7.2.5.
- Match-statements, see Section 7.2.2.
- The curly-brace {} constructor which is used as array constructor in Modelica 3.2 and (de facto) list constructor in MetaModelica 1.0 has been generalized to a `Sequence` constructor that can be used to construct collections of sequence subtypes such as arrays and lists, see Section 6.2.

The following constructs have been deprecated:

- *Removed*: The `subtypeof` keyword is deprecated and replaced by the `constrainedby` keyword which has become part of the Modelica standard.
- *Removed*. The `replaceable type TypeVar subtypeof Any` construct for polymorphic type variables has been deprecated and replaced by the more concise `<TypeVar>` notation, see Section 9.2 and the example below.

For example, the first part of the function `func` with two polymorphic type variables is declared as follows in MetaModelica 1.0 syntax:

```
function func
  replaceable type TypeVar1 subtypeof Any;
  replaceable type TypeVar2 subtypeof Any;
```

This corresponds to a single line in MetaModelica 2.0, shown below:

```
function func<TypeVar1,TypeVar2>
```

5.2.1 Changes to MetaModelica Builtin Types

The MetaModelica language supports a builtin set of primitive data types as well as means of declaring more complex types and structures such as tuples and lists. Below there is a comparison between MetaModelica 1.0 and 2.0 in this respect.

Table 5-4. Builtin parameterized types with constructors in MetaModelica 1.0.

Type Notation Example	*Data Constructor Example*
`tuple<T1,T2,T3>`	`(1,2,3)  tuple(1,2,3)`
`list<T1>`	`{1,2,3}  list(1,2,3)  1::2::3::{}`
`array<T1>  or  T1[:]`	`listArray({1,2,3}),arrayCreate(1,3), arrayUpdate(arr, indices),` etc. This kind of array is the same as `MRArray` in MetaModelica 2.0.
`Option<T1>`	`SOME(1), NONE()`

Table 5-5. Builtin parameterized types with constructors in MetaModelica 2.0.

Type Notation Example	*Data Constructor Example and other Operations*
`Tuple<T1,T2,T3>`	`(1,2,3)  Tuple(1,2,3)`
`List<T1>`	`{1,2,3}  List(1,2,3)  1::2::3::{}`
`MRArray<T1>`	Array constructor `MRArray(1,2,3)` analogous to `{1,2,3}`, `listMRArray({1,2,3})`, and `mrarrayCreate(3,2.1)` Element indexing: `mrarrayGet(arr,indices)` or

	`arr[indices]` Element update: `mrarrayUpdate(arr, indices)` For more information see Section 5.3.
`T1[:],T1[n],T1[:,:], …`	Standard Modelica array constructor: `{}` or `array(…)`
`Option<T1>`	`SOME(1), NONE()`

5.3 MRArray – Globally Mutable Ragged Arrays vs Standard Arrays

MetaModelica introduces a new array data type called `MRArray`, for globally mutable, possibly ragged, arrays.

5.3.1 Modelica Arrays, Mutability, and Time-Dependent Simulation

Standard Modelica arrays are *locally mutable*, i.e., local array variables and output array variables in functions can be updated at run-time. This means that such array elements can be updated and thus change value at run-time. A local *update* to an array in a function will *not affect* the contents of data structures in other functions or in the simulation state. Such local arrays are not part of the global simulation state. Arrays declared in the top-level scope, i.e., the main function called from the operating system, are regarded as arrays local to that main function and can therefore also be updated. The top-level scope is sometimes called the scripting level.

On the other hand, standard Modelica *arrays in models* during *time-dependent simulation* are *immutable*, i.e., do not change value after they have obtained their value. All variables and array elements during time-dependent simulation are functions of time. When a variable is assigned in a statement in an algorithm section outside of a function this contributes to solving the value of that value at the current point in time; it does *not affect* the variable value at previous points in time. Once an unknown variable has been solved for, it does not change value, i.e., it is immutable.

5.3.2 The MRArray Data Type

The `MRArray` (mutable array) data type available in MetaModelica 2.0 is the same globally mutable array data type that was available already in MetaModelica 1.0, but under a different name. It has the following properties:

- These arrays are globally mutable. This means that such an array can be passed to a function and updated, thereby immediately changing its array value globally available, i.e., also in other functions that might reference the same array object.
- They may be used to represent ragged arrays, i.e., arrays of arrays where the sub-arrays may have different lengths. This is different from standard Modelica arrays which are rectangular, i.e., all sub-arrays always have the same size.
- Modelica array slice operations are not allowed on these arrays.
- These arrays may not be part of the simulation state, and can therefore only be used in functions, not in equation-based models. One reason is that they can be allocated dynamically at run-time. The number of elements may thus not be known until run-time, which would make it impossible for the solver to know about the number of variables.

The following operations are available for globally mutable arrays. For detailed explanations, see Appendix B, Section B.3. Conversion between arrays, lists, and MRArrays are explained in more detail in Section 6.5.

Table 5-6. Operations on globally mutable arrays, `MRArray`, in MetaModelica 2.0 vs MetaModelica 1.0.

Array Operation in	*Array Operation in*	*Explanation*

MetaModelica 2.0	*MetaModelica* 1.0	
`MRArray<T1>`	`array<T1> or T1[:]`	Declaration, 1D array.
`MRArray<MRArray<T1>>`	`array<array<T1>>`	Declaration, 2D array, i.e., array of array.
`MRArray(v1,v2,v3,…)`	Not available. Instead `listArray` was used.	Constructor. Creates an 1D array with element values v1,v2,v3, etc.
`mrarrayGet(arr,ind)`	`arrayGet(arr,ind)`	Indexing, can also use `arr[ind]`.
`mrarrayCreate(n,v)`	`arrayCreate(n,v)`	Creates a new array filled with value v.
`mrarrayLength(arr)`	`arrayLength(arr)`	Returns number of elements in `arr`.
`mrarrayNList(arr)`	Not available.	Converts possibly nested array `arr` recursively to a list of its elements.
`mrarrayList(arr)`	`arrayList(arr)`	Converts `arr` to a list of its elements.
`listNMRArray(lst)`	Not available.	Converts possibly nested list `lst` recursively to an `MRArray` of its elements.
`listMRArray(lst)`	`listArray(lst)`	Converts `lst` to an `MRArray` of its elements.
`mrarrayUpdate(arr,i,el)`	`arrayUpdate(arr,i,el)`	Updates element in `MRArray arr` to value `e1`. The syntax `arr[i]:=` is not allowed for MRArrays.
`mrarrayCopy(arr)`	`arrayCopy(arr)`	Create a copy of array arr.
`mrarrayAdd(arr,el)`	`arrayAdd(arr,el)`	Create a copy of array arr with element e1 added at the end.

Chapter 6

Collection Types and Operations

Collection types such as arrays or lists are typically used in symbolic processing to represent sequences of elements. These data type are parameterized in terms of the type of the element type they hold.

6.1 Parameterized Data Types

A *parameterized data type* in MetaModelica is a type that may have another type as a parameter. A parameterized type available in most programming languages is the array type which is usually parameterized in terms of its array element type. For example, we can have integer arrays, string arrays, or real arrays, etc. depending on the type of the array elements. The size of an array may also be regarded as a parameter of the array.

The MetaModelica language provides three basic kinds of parameterized types as well as user-defined parameterized union types and record types.

- Lists – the `List` builtin predefined type constructor, parameterized in terms of the type of the list elements. For example, the type of a list of integers is denoted `List<Integer>`, see Section 6.3.
- Arrays – parameterized in terms of the array element type, see Section 6.4.
- Option types – the `Option` builtin predefined type constructor, parameterized in terms of the type of the optional value. An example of an option type is `Option<String>`, see Section 7.1.3.
- Parameterized user-defined union types and record types, see Section 7.1.2.1.

Note that all basic parameterized types in MetaModelica are *monomorphic*: all elements have to have the same type, i.e., you cannot mix elements of type `Real` and type `String` within the same array or list. Certain languages provide *polymorphic* arrays, i.e., array elements may have different types.

However, arrays of elements of "different" types in MetaModelica can be represented by arrays of elements of union types (Section 7.1.2), where each "type" in the union type is denoted by a different record constructor.

6.2 Sequence Type and { } Constructor

Arrays and lists are special cases of monomorphic sequences, i.e, ordered collections of elements, all of the same type. Therefore we introduce an abstract type called `Sequence`. Both the `List` type and the `MRArray` (mutable ragged array, Section 5.3) type are subtypes of `Sequence`, i.e., specialized types compatible with `Sequence` but with additional properties. The standard Modelica rectangular array type (`Array`) is a subtype of the ragged array type (`RArray`). However, only `List`, `Array`, and `MRArray` are available in MetaModelica 2.0. See Figure 6-1 below.

Collections of `Sequence` type are vectorizable.

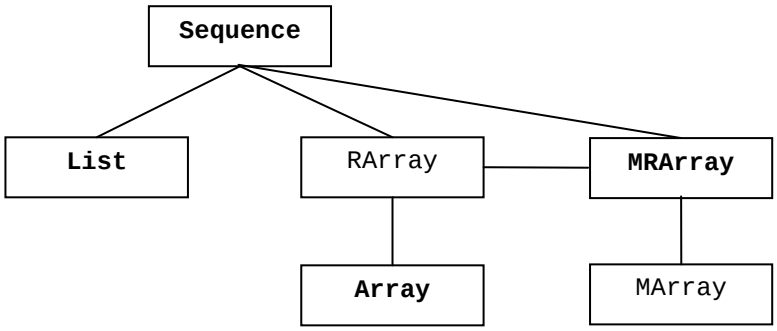

Figure 6-1. A type hierarchy of sequence types. The abstract `Sequence` type is at the top of the hierarchy. `List` , `RArray` (ragged arrays), and `MRArray` (globally mutable ragged arrays) are subtypes of `Sequence`, i.e., more specialized variants with sequence properties. The array types `Array` (locally mutable) and `MRArray` (globally mutable) are rectangular arrays. The standard Modelica rectangular `Array` is a subtype of `RArray`. Only `List`, `Array`, and `MRArray` are available in MetaModelica 2.0. The `Sequence` type is only available as an abstract type, i.e., cannot be used to declare variables.

There is a need for a readable and concise notation for literal constants and creation of sequences of all kinds. Some languages use specialized syntax for each kind of constructor, e.g., the curly-brace array constructor {} used for standard rectangular arrays in Modelica, and square-brackets `[]` as a notation for list construction. Using square-brackets `[]` for lists is not possible in Modelica since `[]` is an operator for the construction and concatenation of square matrices.

Another alternative of always requiring the `List()` constructor to create lists, `MRArray()`to create mutable ragged arrays, and `Array()`to create rectangular arrays would lead to clumsy and less readable notation in many cases.

As a general solution to this problem we abstract and generalize the curly-brace constructor {} to be a constructor for the general `Sequence` type. Since `Sequence` (currently) is an abstract type, the sequence {} constructor will be converted to the specific constructor for each subtype, `Array`, `MRArray`, or `List`, depending on the usage context.

For example, this makes it possible to use the curly-brace constructor {} in most cases as if it was an array constructor or a list constructor. If there is any ambiguity, explicit constructors such as `List(…)`can be used. See also Section 9.5.1 regarding implicit conversions and Section 6.5 for a complete table of the conversion operators between arrays, lists, and conversions from sequences.

6.3 Lists

Lists are common data structures in declarative languages since they conveniently allow representation and manipulation of sequences of elements. Elements can be efficiently (in constant time) added to beginning of lists in a declarative way. The list data structure described here is immutable, i.e., a list once created is never modified, and can therefore easily be reused as part of other lists or complex data structures. The following basic list construction operators are available:

- The *list constructor*: `{el1,el2,el3,...}` or `List(el1,el2,el3,...)` creates a list of elements `el1`, `el2`, ... of identical type. Examples: `{2,3,4}` and `List(2,3,4)` are lists of integers, `{"aa","b"}` and `List("foo")` lists of strings, etc. For the difference between {…} and List(…) see Sections 6.2 and 9.5.
- The *empty list* is denoted by `{}` or `List()`.

- The *list element concatenation* operation `cons(element, lst)` or the equivalent `::` operator syntax as in `element :: lst`, adds an element in front of the list `lst` and returns the resulting list. For example:

```
cons("a", {"b"})  => {"a", "b"};
cons("a",{})      => {"a"}
"a"::"b"::"c"::{} => {"a","b","c"};
"a"::{"b","c"}    => {"a","b","c"}
```

Additional builtin MetaModelica list operations are briefly described by the following examples; see Appendix B.3.7 on page 193 for type signatures of these functions:

- `listAppend({2,3},{4,5})  => {2,3,4,5}`
- `listReverse({2,3,4,5})   => {5,4,3,2}`
- `listLength({2,3,4,5})    => 4`
- `listMember(3, {2,3,4,5}) => true`
- `listGet({2,3,4,5}, 4)    => 5     // First list element is numbered 1`
- `listDelete({2,3,4,5},2)  => {2,4,5}`

The most readable and convenient way of accessing elements in an existing list or constructing new lists is through pattern matching operations, see Section 7.2.

6.3.1 List Types

The types of lists often need to be specified. Named list types can be declared using `type` declarations:

```
type IntegerList  = List<Integer>;
```

An example of a list type for lists of `Real` elements:

```
type RealList     = List<Real>;
```

The following is a parameterized list type with an unspecified element type `Type_elemtype` which is a type parameter (type variable) of the list. Type variables or type names can be referenced within angle brackets `< >`:

```
type ElemList = List<Type_elemtype>;
```

Lists are monomorphic, i.e., all elements must have the same type. Lists of elements with "different" types can be represented by lists of elements of union types, see Chapter 7, where each type in the union type has a different constructor.

6.3.2 List Literals

List literals can be expressed using the list constructor `List(...)`or by the sequence constructor { } which creates a sequence that is automatically converted to a list. For example, the following are one-dimensional list constants:

```
List(1,2,3),     List(3.14, 58E-6)
```

The { } constructor can be used as a list constructor since it constructs a sequence that is automatically converted to a list depending on the type context, e.g. if it is passed to a function formal parameter of list type, or assigned to a variable of list type. In fact, the compiler will usually internally automatically replace the { } constructor with a `List`() constructor.

```
{1,2,3},     {3.14, 58E-6}
```

6.4 Arrays

As already mentioned in Section 5.3, two kinds of arrays are available in MetaModelica: *standard Modelica arrays* as well as *globally mutable ragged arrays* of the `MRArray` data type.

A one-dimensional array is a sequence of elements, all of the same type. The main advantage of an array compared to a list is that an arbitrary element of an array can be accessed in constant time by an array indexing operation on an array using an integer to denote the ordinal position of the accessed element. See Appendix B.3.8 for a full description of builtin array operations for `MRArray`.

The main advantage of a list compared to an array is that an element can be added to the front of the list in constant time without modifying the previous list, i.e., lists are immutable which makes them easy to share in data structures.

Assume that we have a standard Modelica 1D array `vec`, as well as an array of type `MRArray`, both containing `Integer` elements:

```
Integer[:]      vec = {2,4,6,8,10};
MRArray<Integer> mvec;
```

There are several ways of initializing an array variable of type `MRArray`. One method is to use the {} sequence constructor which will be automatically converted to an `MRArray` constructor when assigned to an `MRArray` variable:

```
mvec := {2,4,6,8};
```

Another clumsier variant is to pass the sequence constructed by { } to `listMarray`. The sequence is first converted to a list (since a `List` argument is expected) and then to an `MRArray`. However, the compiler will be smart enough to construct the `MRArray` directly. For example:

```
mvec := listMRArray({2,4,6,8})
```

Accessing the third element of the array `mvec` using the array indexing operation `mrarrayGet`, where the first element has index 1:

```
mrarrayGet(mvec,3) => 6
```

It is also possible to use the more concise square bracket indexing notation to get array element values:

```
mvec[3] => 6
```

Getting the length of the array `mvec`:

```
mrarrayLength(mvec) => 4
```

Getting the length of the standard Modelica array `vec`:

```
size(vec,1) => 5
```

Creating standard arrays of certain length filled with a single element value is possible using fill:

```
fill(3.14,98)   // Gives an array of 98 Real-valued elements with the value 3.14
```

The corresponding for MRArrays:

```
mrarrayCreate(98,3.14)  // Gives an MRArray with 98 Real-valued 3.14 elements
```

Named array types can of course be declared using the **`type`** construct, e.g. as in the declaration of a one-dimensional array of Boolean values:

```
type OneDimBooleanVector1 = Boolean[:];        // Standard arrays
type OneDimBooleanVector2 = MRArray<Boolean>;   // MRArrays
```

Multi-dimensional arrays are represented by arrays of arrays, e.g. as in the following declarations of a two-dimensional matrices of real elements.

```
type OneDimRealVector1 = Real[:];
type TwoDimRealMatrix1 = OneDimRealVector[:];

type OneDimRealVector2 = MRArray<Real>;
type TwoDimRealMatrix2 = MRArray<OneDimRealVector>;
```

Below we give the type signatures, i.e., the types, of input parameters and output results, for a few builtin array operations, also presented in Appendix B.3.8. The following are the length and indexing signatures:

```
function mrarrayLength<Type_a>  "Compute the length of an MRArray"
  input MRArray<Type_a>  inVec;
  output Integer        outLength;
end mrarrayLength;

function mrarrayGet<Type_a> "Extract (indexed access) an array element
                            from the array. Index starts at 1"
  input MRArray<Type_a>  inVec;
  input Integer         index;
  output Type_a         outElement;
end mrarrayGet;
```

The following are signatures of the conversion operations between possibly nested MRArrays and lists:

```
function mrarrayNList<Type_a>  "convert from MRArray to List"
  input  MRArray<Type_a> inVec;
  output List<Type_a>   outLst;
end mrarrayList;

function listNMRArray<Type_a>  "Convert from List to MRArray"
  input  List<Type_a>  inLst;
  output MRArray<Type_a> outVec;
end listMRArray;
```

6.5 Conversions between Arrays, MRArrays, Lists, and Sequences

Below, in Table 5-1, the conversion operators between arrays, MRArrays, and lists, and conversions from objects of Sequence type are briefly presented. See also conversions at assignments or parameter passing, Section 9.5. If the constructor of the input argument is available, the MetaModelica compiler has the opportunity to change the constructor to directly construct the desired data and avoid performing a conversion at run-time.

Table 6-1. Conversion operators between arrays, MRArrays, and lists and from Sequence objects.

Conversion Operator	*Explanation*
`arrayNList(arr)`	Converts possibly nested standard array `arr` to a list of its elements. For arrays of arrays, the conversion is done recursively for elements of array type until non-arrays are encountered.
`arrayList(arr)`	Converts standard array `arr` to a list of its elements. The conversion is not applied recursively to the elements.
`listNArray(lst)`	Converts possibly nested list `lst` to a standard array of its elements. For lists of lists, the conversion is done recursively for elements of list type until non-lists are encountered. All sub-lists at each converted level must have equal length. Sub-lists of unequal length generate an error.
`listArray(lst)`	Converts list `lst` to a standard array of its elements. The conversion is not applied recursively to the elements.
`mrarrayNList(marr)`	Converts possibly nested `MRArray` `marr` to a list of its elements. For MRArrays of MRArrays, the conversion is done recursively for elements of `MRArray` type, until non-MRArrays are encountered.
`mrarrayList(marr)`	Converts possibly nested `MRArray` `marr` to a list of its elements. The conversion is not applied recursively to the elements.
`listNMRArray(lst)`	Converts possibly nested list `lst` to an `MRArray` of its elements. For

	lists of lists, the conversion is done recursively for elements of list type until non-lists are encountered.
`listMRArray(lst)`	Converts list `lst` to an `MRArray` of its elements. The conversion is not applied recursively to the elements.
`arrayNMRArray(arr)`	Converts possibly nested standard array `arr` to an `MRArray`. For arrays of arrays, the conversion is done recursively for elements of array type until non-arrays are encountered.
`arrayMRArray(arr)`	Converts standard array `arr` to an `MRArray`. The conversion is not applied recursively to the elements.
`mrarrayNArray(marr)`	Converts possibly nested `MRArray marr` to a standard array. For MRArrays of MRArrays, the conversion is done recursively for elements of `MRArray` type until non-Marrays are encountered. All sub-MRArrays at each converted level must have equal length. Sub-MRArrays of unequal length generate an error.
`mrarrayArray(marr)`	Converts `MRArray marr` to a standard array. The conversion is not applied recursively to the elements.
`sequenceNList(seq)`	Converts possibly nested `Sequence seq` to a list of its elements. For sequences of sequences, the conversion is done recursively for elements of `Sequence` type, until non-`Sequence` objects are encountered.
`sequenceNArray(seq)`	Converts possibly nested sequence `seq` to a standard array of its elements. For sequences of sequences, the conversion is done recursively for elements of `Sequence` type until non-Sequence objects are encountered. All sub-sequences at each converted level must have equal length. Sub-sequences of unequal length generate an error.
`sequenceNMRArray(seq)`	Converts possibly nested sequence `seq` to an `MRArray` of its elements. For sequences of sequences, the conversion is done recursively for elements of Sequence type until objects of non-`Sequence` type are encountered.

6.6 For-loops, List and Array Comprehensions with Guards

Operations on collections such as arrays or lists include the following constructs:

- Match expressions – for picking elements from collections and transforming them to new expressions, Section 7.2.
- For-loops that iterate over collections, Section 6.6.1.
- Array- and list-comprehensions that iterate over collections and return values.

A guard is a Boolean expression that is evaluated at run-time and can be used to determine if a case-branch should be taken (in match-expressions, Section 7.2) or which values that should be picked up by a collection constructor such as an array- or list-comprehension.

Guards are useful, e.g., giving more expressive power to express filters to select elements from a collection. They can also be used to generalize the conditions for selecting a case-branch in match-expressions which often improves the code structure, readability, and conciseness. Other languages such as Mathematica (Wolfram 2003) and Scala (Odersky, Spoon, and Venners 2008) also have patterns with optional guards.

```
<guarded-pattern> ::=  <pattern-no-guard> [guard <conditional-expr>]
```

For example, for array and list comprehensions, the evaluated guard `conditional-expr` is used as follows

- When `true`, the collected value is included in the reduction/collected value.
- When `false`, the value is not included in the reduction/collected value.
- When failing, the whole iterator expression fails.

Schematic examples:

list comprehension:

```
{ <expr> for <guarded-pattern> in <list-expr> }
```

array comprehension:

```
{ <expr> for <guarded-pattern> in <array-expr> }
```

Below we give several concrete examples of array and list comprehensions.

Example – collect `NODE` *elements with positive x values.*

The following extracts all numbers from `NODE` elements in `nodeList` with a positive number `x`, using the pattern with guard `NODE(x) guard x > 0`, see also Section 7.3 regarding patterns.

```
{ elem for NODE(x) guard x > 0 in nodeList }
```

Example – collect rational numbers represented by the `RATIONAL` *constructor.*

Extract all rational numbers from a list of Numbers. For patterns such as `RATIONAL(x,y)`, see Section 7.3.

```
{ RATIONAL(x,y) for RATIONAL(x,y) in numbers_expr }
```

Example – collect rational numbers with positive denominator.

```
{ RATIONAL(x,y) for RATIONAL(x,y) guard y > 0 in numbers_expr }
```

The `RATIONAL()` constructor function refers to the `RATIONAL` record in the following uniontype:

```
uniontype Number
 record INT      Integer int;               end INT;
 record RATIONAL Integer dividend, divisor; end RATIONAL;
 record REAL     Real real;                 end REAL;
 record COMPLEX  Real re,im;                end COMPLEX;
end Number;
```

Examples – collecting information from a small book database.

This example is from (Odersky, Spoon, and Venners 2008), page 479, also shown in Modelica-like syntax in (Elmqvist 2010). Here the example has been adapted to the previously presented design of array comprehensions with guards. For the example assume that we have a small registry of books:

```
record Book
  String title;
  String authors[:];
end Book;

constant Book books[:] = {
  Book("Structure and Interpretation of Computer Programs",
       {"Abelson","Sussman"}),
  Book("Principles of Compiler Design",
       {"Aho", "Ullman"}),
  Book("Elements of ML Programming",
       {"Ullman"}) };
```

We might then, for example, like to list all titles:

```
{ book.title for book in books }
```

A natural query would be for a specific title. A guard clause then needs to be added:

```
import Modelica.Utilities.Strings.find;
{ bk for bk guard find(bk.title,"Principles") > 0  in books }
```

Combined with a sequence of iterators, we could search for a specific author:

```
{ b for b in books, a guard find(a, "Ullman") > 0 in b.authors }
```

Searching for all authors who have written at least two books could be done as follows:

```
{ a1 for
    b1 in books,
    b2 guard b1<>b2 in books,
    a1 in b1.authors,
    a2 guard a1==a2 in b2.authors }
```

6.6.1 Guarded Patterns in for-loops

Similar to array and list comprehensions, the iteration variable in for-loops can be generalized to a pattern expression including an optional Boolean guard expression. For-loops are generalized to iterate over lists as well as over arrays.

Below is an example of a schematic for-loop iterating over a list and computing an array x:

```
i := 1;
for <elem> in <list-expr> loop
  x[i] := func(<elem>)
  i := i+1;
end loop;
```

This schema can be generalized to using a guarded pattern:

```
for <guarded-pattern> in <list-expr> loop
  x[i] := func(<elem>)
  i := i+1;
end loop;
```

The following is an example of printing positive numbers from an array:

```
for x guard x>0 in {1.2, -3.14, 5.5, -99.0} loop
  print(x);
end loop;
```

Chapter 7

Union Types, Options, and Pattern Matching

This chapter introduces structured data types such as tree structures, tuples, and option types which are important to represent and manipulate models.

- Tuples is a light-weight version of records without field names that is sometimes more convenient to use than full record types.
- Union types are used for tree and graphic data structures, typically to represent abstract syntax trees.
- Option types provide a type-safe way of representing the common situation where a data item is optionally present in a data structure.

7.1 Structured Data Types

7.1.1 Tuples

Tuples are a form of records without field names. They are represented by parenthesized, comma-separated sequences of items each of which may have a different type, e.g.:

- `(55,66)`— a 2-tuple of integers. (referred to as `tup1` below)
- `(55,"Hello",INTconst(77))`— a 3-tuple of integer, `string`, and `Exp`.
- Tuple unnamed field values can be accessed by dot notation, using the ordinal number of the field starting with 1 for the first field.. For example, the above tuple, `tup1`, `tup1.1` is 55, and `tup1.2` is 66.

Named tuple types can be declared explicitly through the `type` declaration using the `Tuple` type constructor:

```
type  TwoInt     = Tuple<Integer,Integer>;
type  Threetuple = Tuple<Integer,String,Exp>;
```

7.1.2 Union Types for Records, Trees, and Graphs

The **`uniontype`** declaration in MetaModelica is used to introduce *union types*, for example the type `Number` below, which can be used to represent several kinds of number types such as integers, rational numbers, real, and complex within the same type:

```
uniontype Number
  record INT       Integer int;                   end INT;
  record RATIONAL  Integer int1;  Integer int2;   end RATIONAL;
  record REAL      Real re;                       end REAL;
  record COMPLEX   Real re;  Real im;             end COMPLEX;
end Number;
```

This syntax is ispired by the corresponding constructs used for algebraic data types in functional programming languages.

The different names, `INT`, `RATIONAL`, `REAL` and `COMPLEX`, are called *constructors*, as they are used to *construct* tagged instances of the type. For example, we can construct a `Number` instance `REAL(3.14)` to hold a real number or another instance `COMPLEX(2.1,3.5)` to hold a complex number.

Each variant of such a union type is actually a *record type* with one or more fields that (currently) can only be referred to by their position in the record. The type `Number` can be viewed as the union of the record types `INT`, `RATIONAL`, `REAL` and `COMPLEX`.

The most frequent use of union types in MetaModelica is to specify abstract syntax tree representations used in language specifications as we have seen many examples of in earlier chapters of this text, e.g. `Exp` below, first presented in Section 2.1.2:

```
uniontype Exp
  record INTconst Integer int;          end INTconst;
  record ADDop    Exp exp1;  Exp exp2; end ADDop;
  record SUBop    Exp exp1;  Exp exp2; end SUBop;
  record MULop    Exp exp1;  Exp exp2; end MULop;
  record DIVop    Exp exp1;  Exp exp2; end DIVop;
  record NEGop    Exp exp;             end NEGop;
end Exp;
```

The record constructors `INTconst`, `ADDop`, `SUBop`, etc. are can be used to construct nodes in abstract syntax trees such as `INTconst(55)` and `ADDop(INTconst(6),INTconst(44))`, etc.

Representing DAG (Directed Acyclic Graph) structures is no problem. Just pass the same argument twice or more and the child node will be shared, e.g. when building an addition node using the `ADDop` constructor below:

```
ADDop(x, x)
```

Building circular structures is not possible because of the declarative side-effect free nature of MetaModelica. Once a node has been constructed it cannot be modified to point to itself. Recursive dependencies such as recursive types have to be represented with the aid of some intermediate node.

7.1.2.1 Parameterized Union Types and Record Types

Records and other classes in standard Modelica 3.2 are parameterizable using the replaceable and redeclare constructs. For example, we could make the above `INT` record in Section 7.1.2 parameterized by adding a type variable `IntType`:

```
record INTconst
  replaceable type IntTypeVar constrainedby Any;
  IntType int;
end INT;
```

In Modelica, by using `INT(redeclare IntTypeVar=Integer)` we will get a result which is equivalent to the previous `INT` class with a field `int` of type `Integer`. If instead we would use `INT(redeclare IntTypeVar=LongInt)` the result would be equivalent to:

```
record INT
  LongInt int;
end INT;
```

In MetaModelica we can use the more concise angle-bracket syntax < > to express type variables in parameterized classes. The following short version is equivalent to the above `INT` record with replaceable:

```
record INT<IntTypeVar>
  IntType int;
end INT;
```

It is also possible to parameterize uniontypes. The above `Number` uniontype parameterized by a type variable `IntTypeVar` appears as follows:

```
uniontype Number<IntTypeVar>
  record INT       IntTypeVar int;  end INT;
  record RATIONAL  IntTypeVar int1;  IntTypeVar int2;  end RATIONAL;
  record REAL      Real re;  end REAL;
  record COMPLEX   Real re;  Real im;  end COMPLEX;
end Number;
```

7.1.3 Option Types

Option types have been introduced in MetaModelica to provide a type-safe way of representing the common situation where a data item is optionally present in a data structure – which in language specification applications typically is an abstract syntax tree.

The `Option` type is a predefined parameterized MetaModelica union type with the two constructors `NONE()` and `SOME()`:

```
uniontype Option<Type_a>
  record NONE end NONE;
  record SOME
    Type_a elem;
  end SOME;
end Option;
```

The constant `NONE()` with no arguments automatically belongs to any Option type. A constructor call such as `SOME(x1)` where `x1` has the type `Type_a`, has the type `Option<Type_a>`.

The constructor `NONE()` is used to represent the case where the optional data item (of type `Type_a` in the above example) is not present, whereas the constructor `SOME()` is used when the data item is present in the data structure. One example is the optional return value in return statements, represented as abstract syntax trees, where the `NONE()` constructor is used for the `return;` variant without value, and `SOME(...)` for the `return(`*valueexpression*`);` variant.

7.2 Pattern-Matching and Match-Expressions

Pattern-matching on instances of structured data types is one of the central facilities provided by MetaModelica, which significantly contributes to the elegance and ease with which many language aspects may be specified. The pattern matching provided by the match-expression construct in MetaModelica is very close to similar facilities in many functional languages and in Scala, but is also related to switch statements in C or Java. It has two important advantages over traditional switch statements:

- A match-expression can appear in any of the three Modelica contexts: expressions, statements, or in equations.
- The selection in the case branches is based on pattern matching, which reduces to equality testing or switch in simple cases, but is much more powerful in the general case.

A very simple example of a match-expression is the following code fragment, which returns a number corresponding to a given input string. The pattern matching is very simple – just compare the string value of `s` with one of the constant pattern strings `"one"`, `"two"` or `"three"`, and if none of these matches return 0 since the wildcard pattern `_` (underscore) matches anything.

```
  String s;
  Real   x;
algorithm
  x := match s
    case "one"   then 1;
    case "two"   then 2;
    case "three" then 3;
    case  _      then 0;
  end match;
```

Alternatively, an else-branch can be used instead of the last wildcard pattern:

```
  String s;
  Real   x;
algorithm
  x := match s
    case "one"   then 1;
    case "two"   then 2;
    case "three" then 3;
    else              0;
  end match;
```

or using a match-expression with the `matchcontinue` keyword:

```
  Real   x;
algorithm
  x := matchcontinue s
    case "one"   then 1;
    case "two"   then 2;
    case "three" then 3;
    else              0;
  end matchcontinue;
```

These are trivial special cases. The general structure and evaluation of match-expressions is described in the following sections.

7.2.1 Syntactic Structure of Match-Expressions

The general structure of match-expressions starting with either the `matchcontinue` or the `match` keyword is indicated by the outline below. See Appendix A for the grammar.

The else-branch is optional and is identical to a `case _` branch. First we show the variant using the `match` keyword:

```
match <m-expr>  <opt-local-decl>
    ...
  case <pat-expr>
    equation
      <opt-equations>
    then <expr>;
    ...
  case <pat-expr>
    equation
      <opt-equations>
    then <expr>;
    ...
  else
    equation
      <opt-equations>
    then <expr>;

end match;
```

The variant with the `matchcontinue` keyword appears as follows:

```
matchcontinue <m-expr>  <opt-local-decl>
    ...
  case <pat-expr>
    equation
      <opt-equations>
    then <expr>;
    ...
  case <pat-expr>
    equation
      <opt-equations>
    then <expr>;
    ...
```

```
  else
    equation
      <opt-equations>
    then <expr>;

end matchcontinue;
```

7.2.1.1 Match-Expressions with Local Algorithm Sections

Match-expressions may have local algorithm sections instead of local equation sections, e.g. as in the following skeleton. Local algorithm sections may contain ordinary Modelica statements, but may *only* update local variables declared after the `local` keyword in the match-expression. That restriction follows from the fact that Modelica expressions are assumed to be pure and not have side-effects. The following example includes two cases with a local algorithm section and one case with a local equation section.

```
match <m-expr>  <opt-local-decl>
    ...
  case <pat-expr>
    algorithm
      <opt-statements>
    then <expr>;
    ...
  case <pat-expr>
    equation
      <opt-equations>
    then <expr>;
    ...
  else
    algorithm
      <opt-statements>
    then <expr>;

end match;
```

7.2.2 Match-Statements

The match-statement is strongly related to the match-expression, but with the following differences:

- Match-statements can be used in statement context in algorithm sections, whereas match-expression can only be used in expression context.
- Match-statements return no value.
- Match-statements allow assignments to variables declared outside the match-statement.

The syntax is as follows:

```
match <m-expr>  <opt-local-decl>
    ...
  case <pat-expr>
    algorithm
      <opt-statements>
    ...
  case <pat-expr>
    algorithm
      <opt-statements>
    ...
  else
    algorithm
      <opt-statements>

end match;
```

The following is an example of a match-statement in the function `printValue`. Note that variables (such as `vstr`) declared outside the match-statement can be assigned to, which is not allowed in a match-expression. The then-parts of the cases are not present since no value is returned from a match-statement.

```
function printValue  "A function with a match-statement)
  input Env.Value inValue;
protected
  String vstr;
algorithm
  match inValue
    local
      Integer v;
      Real r;
    case Env.INTVAL(integer = v)
      algorithm
        vstr := intString(v);
        print(vstr);
        print("\n");
    case Env.REALVAL(real = r)
      algorithm
        vstr := realString(r);
        print(vstr);
        print("\n");
    case Env.BOOLVAL(boolean = true)
      algorithm
        print("true\n");
    case Env.BOOLVAL(boolean = false)
      algorithm
        print("false\n")
  end match;
end printValue;
```

The same function is shown below, but using a match-expression. Since match-expressions are declarative, only local variables can be assigned.

```
function printValue
  "A function with a match-expression
   returning the empty tuple value"
  input Env.Value inValue;
algorithm
  _ := match inValue
    local
      String vstr;
      Integer v;
      Real r;

    case Env.INTVAL(integer = v)
      algorithm
        vstr := intString(v);
        print(vstr);
        print("\n");
      then ();

    case Env.REALVAL(real = r)
      algorithm
        vstr := realString(r);
        print(vstr);
        print("\n");
      then ();

    case Env.BOOLVAL(boolean = true)
      algorithm
        print("true\n");
      then ();
```

```
    case Env.BOOLVAL(boolean = false)
      algorithm
        print("false\n");
      then ();
  end match;
end printValue;
```

7.2.3 Evaluation of Match-Expressions and Match-Statements

Match-expressions, and match-statements when applicable, have the following semantic properties:

- The *match data value* computed by the expression `<m-expr>` (see previous section) is matched against the *patterns* `<pat-expr>` occurring in each case-branch. In the simple case, matching is just equality testing of the match data value against a constant pattern. In the general case the matching is performed by a unification algorithm which may assign unbound pattern variables (declared in local declarations starting with the `local` keyword) to values during the matching process.
- The *match data value* computed by the expression `<m-expr>` can be of simple type, `List` type, union type, or record type, but currently not array type or `MRArray` type.
- `match`. The match data value is matched against each of the patterns after the `case` keywords in order; if one match fails or the match succeeds but the guard expression is false, the next is tried until there are no more case-branches in which case (if present) the else-branch is executed. *If a matching against a pattern succeeds and the guard expression, if present, is true but the rest of the computation in that case-branch fails, then the whole match-expression immediately fails.*
- `matchcontinue`. The match data value is matched against the patterns in the case-branches in the order they are declared. If the match against a pattern succeeds and the guard expression, if present, is true, the rest of the case-branch is evaluated. If that evaluation fails the computation continues with the next case-branch. If the match against a pattern fails or the match succeeds but *the rest of the computation in that case-branch fails* , the matching *continues* with the next case-branch. If none of the cases succeed, the else-branch is evaluated if present. If all case-branches fail and the else-branch fails (if present), the whole match-expression fails.
- If an equation or an expression in a case-branch of a match-expression fails, all local variables become unbound, and matching continues with the next branch.
- Only algebraic equations are allowed as local equations, no differential equations.
- Only locally declared variables (local unknowns) declared by local declarations within the match-expression are solved for. Only such local variables may appear as pattern variables.
- Local equations are solved in the order they are declared. *Restriction*: unbound local variables to be solved for may appear only on the left-hand side of local equations, not on the right-hand side; for example: `x = false`; (OK), but `false=x;` (not OK). See also Section 7.4.
- The scope of local variables (after the `local` keyword in match-expressions) declared at the *top* of a match-expression extends throughout the rest of the whole match-expression.

7.2.4 Forms of Patterns

Patterns can have the following forms:

- Patterns can contain calls to record constructor functions, *not* to other kinds of functions.
- Positional and/or named argument constructor call syntax can be used in patterns containing constructors, e.g. the positional call `FOO(1,_,2)` is allowed (Section 7.2.4.3); a named argument call version, e.g., `FOO(field1=1,field3=myvar)`, is also allowed (Section 7.2.4.4).

- A constructor pattern can have an unspecified argument list denoted by double underscore __, as in `FOO(__)`. This matches the corresponding constructor (here `FOO`) with arbitrary arguments. For an example see Section 7.2.5.
- Patterns can contain curly-brace array and `List` constructors, e.g., `{3,5,...}`, `List("a","b")`, see Section 6.2, and tuple constructors e.g. (...,...,...), see Section 7.1.1.
- Patterns may currently (in MetaModelica 2.0) not have any array constructor.
- Patterns can contain literal constants, e.g. `"string2"`, 3.14, 555, `true`, `false`.
- Patterns can contain the _ wildcard which matches anything.
- Patterns can contain the **`as`** binding operator, e.g. `state1 as (env,_,_)`, see Section 2.5.4.3 for an example.
- Patterns can contain the `::` operator, e.g. as in: `((id2,value)::_ ,id)`, see also Section 6.2.
- Patterns may optionally have guards (Section 7.2.4.1), i.e., conditional expressions that are evaluated at run-time.
- Only local variables declared in local declarations after the `local` keyword may appear as unbound pattern variables in patterns.

7.2.4.1 Guarded Patterns

A pattern also includes an optional Boolean conditional expression preceded by the `guard` keyword as specified by the following grammar rule for a pattern expression:

```
<pat-expr> ::=  <pattern-no-guard> [guard <bool-expr>]
```

The matching process of such a pattern first performs structural matching of the part without guard, the `<pattern-no-guard>`. If that is successful, the guard conditional expression, `<bool-expr>`, is evaluated. If it evaluates to `true`, the whole pattern is considered matching successfully, otherwise the matching fails.

For example, regard the following two cases with guarded patterns obtained from the function `numberAbs` below:

```
case REAL(x) guard x >  0 then  x;
case REAL(x) guard x <= 0 then -x;
```

Given a constructor `REAL(-3.14)` containing a Real number -3.14. This is passed in the call `numberAbs(REAL(-3.14))`. The first case will match OK structurally but the guard expression `x>0` will evaluate to `false` which makes the matching fail. Therefore the next case is tried which matches structurally and also has a guard expression `x<=0` which evaluates to `true`, making the match succeed.

```
function numberAbs "Returns the absolute value of a Number as a Real."
  input Number number;
  output Real result;
algorithm
  result := match number
    local Integer int,dividend,divisor; Real x,re,im;
    case INT(int) then abs(int);
    case RATIONAL(dividend,divisor) then abs(dividend/divisor);
    case REAL(x) guard x>0 then x;
    case REAL(x) guard x<=0 then -x;
    case COMPLEX(im,re) then abs(sqrt(im^2+re^2));
  end match;
end numberAbs;
```

7.2.4.2 Patterns with the as Binding Operator

The resulting value of a match of a pattern (also possible for pattern subexpressions) can be named, and bound to a local variable using the `as` binding operator. The structure of such a pattern subexpression is the following:

```
newname as pattern-expression
```

This means that if `pattern-expression` successfully matches a value, a variable with the name `newname` and having a type equivalent to the type of `pattern-expression`, is bound to that value.

The following example from the Assignments language in Section 2.5.4.3, shows the use of the `as` binding operator:

```
case (state as (env,_,_), WHILE(comp,s1)) // while false ...
  equation
    BOOLval(false) = eval(env,comp);
  then state;

case (state as (env,_,_), WHILE(comp,s1)) // while true ...
  equation
    BOOLval(true) = eval(env,comp);
    state2 = evalStmt(state,s1);
    state3 = evalStmt(state2,WHILE(comp,s1);
  then state3;
```

7.2.4.3 Positional Argument Constructor Call Patterns

Positional constructor call syntax can be used in patterns containing constructors, e.g. positional call patterns such as `FOO(1,_,2)` or `ADDop(e1,e2)` are allowed.

The following function `eval`, from Section 2.1.4.2, shows an example of the usual positional matching, thereby giving values to the initially unbound pattern variables `e1` and `e2`, e.g. in patterns such as `ADDop(e1,e2)`.

```
function eval
  input  Exp     inExp;
  output Integer outInteger;
algorithm
 outInteger := match inExp
    local Exp e1,e2; Integer v1;
    case INTconst(v1) then v1;
    case ADDop(e1,e2) then eval(e1) + eval(e2);
    case SUBop(e1,e2) then eval(e1) - eval(e2);
    case MULop(e1,e2) then eval(e1) * eval(e2);
    case DIVop(e1,e2) then eval(e1) / eval(e2);
    case NEGop(e1)    then –eval(e1);
   end match;
end eval;
```

A longer version with local equations but with the same semantics:

```
function eval
  input  Exp     inExp;
  output Integer outInteger;
algorithm
 outInteger := match inExp
    local
      Integer v1;
      Exp     e1,e2;

    case INTconst(v1) then v1;

    case ADDop(e1,e2)
      equation
        v1 = eval(e1; v2 = eval(e2);
      then v1 + v2;

    case SUBop(e1,e2)
      equation
        v1 = eval(e1); v2 = eval(e2);
      then v1 - v2;

    case MULop(e1,e2)
      equation
        v1 = eval(e1); v2 = eval(e2);
```

```
      then v1 * v2;

    case DIVop(e1,e2)
      equation
        v1 = eval(e1); v2 = eval(e2);
      then v1 / v2;

    case NEGop(e1)
      equation
        v1 = eval(e1);
      then –v1;

  end match;
end eval;
```

7.2.4.4 Named Argument Constructor Call Patterns

In addition to the *positional constructor call pattern matching* where the positions of the arguments to the constructor is decisive, it is also possible to use *named pattern matching*, using the record field names of the corresponding record declaration to specify the pattern arguments.

Thus, the pattern `ADDop(e1,e2)` would appear as `ADDop(exp1=e1,exp2=e2)` using named pattern matching. An important *advantage* with named pattern matching is that only the parts of the pattern arguments that participate in the matching need to be specified. The wildcard arguments need not be specified. Another advantage is increased readability since the field names are made visible in the patterns.

We show the corresponding union type for convenience:

```
uniontype Exp
  record  INTconst  Integer int;       end INTconst;
  record  ADDop  Exp exp1;  Exp exp2;  end ADDop;
  record  SUBop  Exp exp1;  Exp exp2;  end SUBop;
  record  MULop  Exp exp1;  Exp exp2;  end MULop;
  record  DIVop  Exp exp1;  Exp exp2;  end DIVop;
  record  NEGop  Exp exp;              end NEGop;
end Exp;
```

Below we have changed all cases in the previous `eval` function example to use named pattern matching:

```
function eval
  input  Exp     inExp;
  output Integer outInteger;
algorithm
 outInteger := match inExp
   local
     Integer v1;
     Exp     e1,e2;

    case INTconst(v1) then v1;

    case ADDop(exp1 = e1, exp2 = e2)
      equation
        v1 = eval(e1); v2 = eval(e2;
      then v1 + v2;

    case SUBop(exp1 = e1, exp2 = e2)
      equation
        v1 = eval(e1); v2 = eval(e2);
      then v1 - v2;

    case MULop(exp1 = e1, exp2 = e2)
      equation
        v1 = eval(e1); v2 = eval(e2);
      then v1 * v2;

    case DIVop(exp1 = e1, exp2 = e2)
```

```
      equation
        v1 = eval(e1); v2 = eval(e2);
      then v1 / v2;

    case NEGop(exp = e1)
      equation
        v1 = eval(e1);
      then –v1;
  end match;
end eval;
```

7.2.5 Compact Matching and Named Field Access via Dot Notation

The long form of named patterns presented in Section 7.2.4.4 is expressed using `fieldname=patvariable`, e.g. as in:

```
case constructor(fieldname1 = patvariable1, fieldname2 = patvariable2, ...) ...
```

Alternatively it is possible to access `x.fieldname1`, `x.fieldname2`, ... through a variable `x` bound to the corresponding constructor record using the as-binding (Section 7.2.4.2) and avoid introducing additional variables like `patvariable1` and `patvariable2`. For example:

```
case x as constructor(__) ...  x.fieldname1 ... x.fieldname2 ...
```

The field names are associated with the bound variable `x`, which makes the code that refers to them more understandable and eliminates the need for some local variables. In order to access the fields through dot-notation the specific record type of x is inferred (Section 9.3) from the as-binding to the record constructor. Thus the compiler can check that `fieldname1`, `fieldname2`, etc., accessed via the variable `x`, are fields in the corresponding constructor record..

In the eval function below we use the compact matching syntax and access of fields via dot-notation. For example, the pattern `MULop(__)` matches the same as `MULop(exp1=e1,exp2=e2)` but without the need to declare `e1` and `e2` as local variables since they can be accessed as `x.exp1` and `x.exp2`.

```
function eval
  input  Exp     inExp;
  output Integer outInteger;
algorithm
  outInteger := match inExp
    local Exp x;
    case x as INTconst(__) then x.INTconst.int;
    case x as ADDop(__) then eval(x.exp1) + eval(x.exp2);
    case x as SUBop(__) then eval(x.exp1) - eval(x.exp2);
    case x as MULop(__) then eval(x.exp1) * eval(x.exp2);
    case x as DIVop(__) then eval(x.exp1) / eval(x.exp2);
    case x as NEGop(__) then –eval(x.exp);
  end match;
end eval;
```

Another possibility is to access the input variable `inExp` directly. One difficulty is that that `inExp` is a uniontype rather than a specific record type with known field names. In order to be able to access the fields of `inExp` through dot-notation, type inference (Section 9.4) can be used to infer the specific record type of `inExp` in the scope of each case, as in the following example.

Within the scope of `case INTconst(__)` the variable `inExp` is inferred to have the record type `Exp.INTconst` whereas within the scope of `case ADDop(__)` the variable `inExp` is inferred to have the record type `Exp.ADDop`, and analogously for the other cases.

```
function eval
  input  Exp     inExp;
  output Integer outInteger;
algorithm
  outInteger := match inExp
    case INTconst(__) then inExp.int;
    case ADDop(__) then eval(inExp.exp1) + eval(inExp.exp2);
```

```
    case SUBop(__) then eval(inExp.exp1) - eval(inExp.exp2);
    case MULop(__) then eval(inExp.exp1) * eval(inExp.exp2);
    case DIVop(__) then eval(inExp.exp1) / eval(inExp.exp2);
    case NEGop(__) then –eval(inExp.exp);
  end match;
end eval;
```

7.3 Usage of Patterns

Patterns can occur after the `case` keyword, before the `in` keyword in array- or list-comprehensions (Section 6.6), after the `for` keyword in for-loops (Section 6.6.1), and on the left-hand side of the = or := sign in equations or statements respectively.

7.3.1 Examples

Regard the pattern `INT(x)` in the case below:

```
match argument
  local Integer x;
case  INT(x) ...
...
```

This means that `argument` is matched using the pattern `INT(x)`. If there is a match, the case is invoked and the local variable `x` is bound to the argument of `INT`, e.g. `x` will be bound to `55` if `argument` is `INT(55)`.

For cases where the value of the pattern variable is not referenced in the rest of the case, an *anonymous pattern* can be used instead. The pattern variable `x` is then replaced by an underscore in the pattern, as in `INT(_)`, to indicate matching of an anonymous value.

Patterns can be nested to arbitrarily complexity and may contain several pattern variables, e.g. `ADD(INT(x), ADD(y,NEG(INT(77))))`. Patterns may also be pure constants, e.g. `55`, `false`, `INT(55)`.

7.3.2 Patterns in Left-hand-sides of Equations or Statements

Patterns may occur on left-hand sides of local equations or statements. For example:

```
match ...
  local  Integer u;  String w;
case ...
  equation
    (u,w) = ...;
```

If the right-hand side of the local equation produces the tuple `(55,"Test)`, and `u` and `w` are unbound, then the match to the pattern `(u,w)` will succeed by binding `u` to 55 and `w` to `"Test"`.

7.3.3 Constraint Equations

An equation may also be used as a constraint. For example, the variable `inExp`, which already has a value, is here constrained to having the value `false`, otherwise the equation will fail:

```
false = inExp;   // inExp is constrained to having the value false
```

Another example is from the translational semantics of PAM in the `transExpr` function in Section 3.1.6.1. Here the equation will be solvable only if `transExpr(e2)` returns a generated code which consists of a simple load instruction, i.e., it is constrained to returning such a result.

```
equation
  List(Mcode.MLOAD(operand2)) = transExpr(e2);
```

Otherwise, the equation will fail and matching will continue with the next case-branch if `matchcontinue` is used, otherwise the whole match-expression fails if `match` is used..

7.3.4 Constructing Values

Values are often created by calling constructors possibly with bound pattern variables supplied as arguments . For example, regard the tuple construction in the case below after the **`then`** keyword:

```
case  ... then (inExp, {5,y}, INT(z))
```

If the case matches and succeeds and `inExp` is already bound to 44, `y` to `"Hello"` and `z` to 77, respectively, then the following tuple term is constructed and returned as the value of the function to which the case belongs:

```
(44, {5,"Hello"}, INT(77))
```

7.4 Forms of Equations in Match Cases

The local equations in a match-expression case are currently restricted to having the following forms, where `funcName` is the name of a function; see also the MetaModelica grammar in Appendix A; `var_or_const` is the name of a variable or a constant such as `false`, etc., or a constant expression, an `expr` may contain constants, variables, constructor calls, operators, and functions

- `expr = funcName(...)`
- `func_name(...)`
- `var_or_const = expr`
- `failure(var_or_const = expr)`
- `failure(func_name(...))`
- `failure(expr = funcName(...))`

The `failure()` operator succeeds if the local equation it operates on fails. Each of these forms can also be parenthesized.

7.4.1 Logically Overlapping Patterns

A programming language specification in MetaModelica is often written in such a way that the local equations of different cases in a function are logically overlapping. For example, the predicates inExp<5 and $3 \leq$ inExp < 10 are logically overlapping since there are values of inExp, in the interval [3,5) that satisfy both predicates.

Below we specify a function `func`, which is specified to return `inExp` + 10 when `inExp` < 5, and `inExp` + 20 for $3 \leq$ `inExp` < 10. This is logically ambiguous in the interval $3 \leq$ `inExp` < 5 where both alternatives are valid.

```
function func
  input Real in_inExp;
  output Real out_y;
algorithm
  out_y := matchcontinue in_inExp
    local Real inExp,y;
    // inExp < 5
    case inExp
      equation
        true = inExp < 5;
      then inExp + 10;
    // inExp >= 3 and inExp < 10
    case inExp
      equation
```

```
        true = (inExp >= 3);
        true = (inExp < 10);
      then inExp+20;
  end matchcontinue;
end func;
```

The determinate search rule of match-expressions in MetaModelica will resolve such ambiguities since the first matching case will always return in the interval $3 \leq$ inExp < 5. Thus, the first case giving the value inExp + 10 will be selected.

There is one rather common case where logically overlapping cases together with MetaModelica's search rule of case matching top-down, left-to-right, can be used to advantage, to allow more *concise* and easily readable specifications. The cases can be ordered such that cases with more *specific* conditions appear first, and more *general* cases which may logically overlap some previous cases appear later.

7.4.2 Default Case in Match-Expressions

There are common situations in specifications where a large number of cases are handled similarly, except a few special cases which need to be treated specially. For example in the function `isunfold` below, where only the `UNFOLD` node returns `true`. All other nodes — which here are mentioned explicitly as separate cases — return `false`.

```
function isunfold
  input  Ty      inNode;
  output Boolean outRes;
algorithm
  outRes := match inNode
      case UNFOLD(_) then true;
      case ARITH(_)  then false;
      case PTR(_)    then false;
      case ARR(_,_)  then false;
      case REC(_)    then false;
    end match;
end isunfold;
```

A more concise specification of this function can be obtained by adding a default case at the end of the match-expression with a general pattern that matches all cases returning the same default result. The top-down, left-to-right search rule in match-expressions ensures that the special cases will match if they occur — before the default case which always matches.

The default pattern consists of a general matching pattern `_` , as in the example below. The default `case _` can be replaced by the `else` keyword which is completely equivalent.

```
function isunfold
  input  Ty      inNode;
  output Boolean outRes;
algorithm
  outRes := match inNode
      case UNFOLD(_) then true;
      case _         then false;
    end match;
end function;
```

Chapter 8

Failures, Exceptions and Error Reporting

There are two ways to generate a failure (also called exception) in MetaModelica

- A failure is generated by some error during the computation, e.g. array index out of bounds[1], division by zero, etc.
- A failure is explicitly generated by calling the MetaModelica `fail()` operator.

A failure (i.e., a kind of exception) is handled in the following way:

- If the failure occurs during evaluation/solution of one of the local equations (or functions called from within such a local equation) in a case of a match-expression, matching is continued in the following case. If all subsequent cases are tried and no one matches and succeeds, the whole match-expression will fail, and in the common case where a match-expression comprise the body of a function, the whole function will fail.
- If the failure occurs during evaluation/solution of one of the statements in a function outside a match-expression, the rest of the function will be aborted and the function will fail.
- Explicit testing for failure of a statement or equation can be done by the `failure(statement/equation)` call. If the `statement` or `equation` fails, then `failure(statement/equation)` succeeds.

See also the discussion of failure versus negation in Section 8.1 and failures and retry in functions, Section 8.2.

Moreover, when an error is reported it is sometimes relevant to give information about the corresponding source code location, see Section 8.3.

8.1 Function Failure Versus Boolean Negation

We have previously mentioned that MetaModelica functions can fail or succeed, whereas conventional functions always succeed in returning some value. The most common cause for an Modelica function to fail is the absence of a case that matches and/or have local equations that succeed.

Another cause of failure is the use of the builtin Modelica command `fail`, which causes a *case* in a match-expression with the `matchcontinue` keyword to fail immediately, subsequently trying the next case that matches, if there is one. On the other hand, a `fail` in a match-expression with the `match` keyword will cause the whole match-expression to fail immediately.

It is important to note that `fail` is quite different from the logical value `false`. A function returning `false` would still succeed since it returns a value. The builtin operator `not` operates on the logical values `true` and `false` according to the following definition:

[1] At the time of this writing the failure behavior in the current implementation is not entirely consistent for certain kinds of run-time faults. For example, index-of-bounds of MArrays generates a failure, whereas inde-out-of-bounds for standard arrays aborts execution. Arithmetic overflow is not detected. Division by zero sometimes generates exceptions. The behavior is not defined by the Modelica standard.

```
function boolNot
  input  Boolean inBool;
  output Boolean outBool;
algorithm
  outBool := if inBool == true then false else true;
end boolNot;
```

However, failure can in a logical sense be regarded as a kind of negation—similar to negation by failure in the Prolog programming language. A local equation that fails will certainly cause the containing case to fail. The MetaModelica `failure()` operator can however invert the logical sense of a proposition. The following local equation is logically successful since it succeeds (but it does not return the predefined value true):

```
failure(function_that_fails(inExp))
```

The two operators `not` and `failure()` thus represent different forms of "negation"—negating the boolean value `true` or `false`, or negating the failure of a call to a function.

8.2 Failure and Retry in Functions with matchcontinue

Two important properties of MetaModelica functions are absent for ordinary functions:

- Functions in MetaModelica can fail or succeed.
- Retry is supported between cases in a match-expression using `matchcontinue`.

A call to a function can *fail* instead of always returning a result which is the case for functions. This is convenient for the specification writer when expressing semantics, since other possibly matching cases in the function will be applied without needing "try-again" mechanisms to be directly encoded into specifications. The failure handling mechanism can also be used in general declarative programming, e.g. the factorial example previously presented in Section 2.3.2.

This brings us into the topic of case retry. If there is a failure in case, or in one of the functions directly or indirectly called via the local equations of the case, and a match-expression is used, MetaModelica will backtrack (i.e., undo) the part of the "execution" which started from this case, and automatically *continue* with the next case (if there is one) in top-down, left-to-right order. If no case in the function matches and succeeds, then the call to this function will fail. Correct back-tracking is however dependent on avoidance of side-effects in the cases of the specification.

8.3 Error Reporting Location Information

In standard Modelica an `assert` equation or statement of the following form is available:

```
assert(condition, message, level = AssertionLevel.error);
```

It can be used, e.g. as in the following example:

```
assert(T > 200 and T < 500, "Medium model outside feasible region");
```

There is also a standard Modelica external C function `ModelicaError` function mentioned in the Modelica specification. In the OpenModelica support libraries the `Error.assert` function is currently available. For example:

```
Error.assert(cond, {"MyFunction.abc failed"});
```

This function only prints `Error: MyFunction.abc failed` without any information about where the error occurred.

In all these cases it would be useful to be able to give more precise information about where the error occurred. Such information is given by the new sourceInfo function below, which will be proposed for standardization:

```
function sourceInfo
```

```
  output SourceInfo;
external "builtin";
end sourceInfo;

record SourceInfo
   String filename,scope;
   Integer lineStart,colStart,lineEnd,colEnd;
end SourceInfo;
```

The following example shows error handing and error reporting with location information:

```
...
case ABC(...)
   equation
     ...
   then ...;
 else
   equation
     source = sourceInfo(); // Location information, line 37 in file abc.mo
     Error.addSourceMessage(Error.INTERNAL_ERROR,
      {"Failed to handle the case: " + ... + " in scope: " + source.scope},
       source);
    then fail();
```

In this case the following error message including location information is produced:

```
[.../abc.mo:37:13-37:25:writable] Error: Internal Error: Failed to handle the
case: DEF(...) in scope Module.printAbcAndDef
```

Chapter 9

Type Inference and Implicit Type Conversion

Type inference is used in MetaModelica in the following cases:

- Declaration equations and declaration assignments where the type can easily be deduced from the right-hand side, Section 9.1.
- Inference of the actual type of polymorphic type variables used in functions that are parameterized by one or more polymorphic types. The inference is based on the types of the actual arguments, e.g. in a function call. See Section 9.2.
- Inference of the specific type of an as-bound variable in case it is declared as a union type, Section 9.3.
- Inference of the specific type of an input variable to pattern matching in the scope of matched pattern in the case where the input variable is declared as a union type, Section 9.4.

Implicit type conversion from Integer to Real occurs in standard Modelica in contexts requiring operations on Real. In MetaModelica there is also implicit type conversion of arrays to lists in contexts requiring list operations, and from arrays and lists to MRArrays, see Section 9.5.

9.1 Types in Declaration Equations and Declaration Assignments

In declaration equations or declaration assignments the type can be inferred locally from the type of the right-hand side expression. For example:

```
Real inExp = 3.15;
```

can be replaced by the following, started by the `var` keyword:

```
var inExp = 3.15;
```

The following is a more verbose example (Elmqvist 2010), where the long name is repeated twice:

```
Modelica_LinearSystems2.Math.Polynomial =
  Modelica_LinearSystems2.Math.Polynomial({1,2,3});
```

The type of p can be inferred from the right hand side, giving:

```
var p = Modelica_LinearSystems2.Math.Polynomial({1,2,3});
```

9.1.1 Potential Problems Using Type Inference

Type inference should be used with care, and only when giving distinct advantages. For example, leaving out the type and instead use type inference sometimes creates problems:

- Leaving out the type information also removes valuable documentation from the program/model. The programmer may have to infer the same information to understand the program, either by hand or using a tool.

- Leaving out type information can have unexpected consequences and cause errors.

For example, regard the following declaration:

```
Real r = 3;
```

Replacing this declaration is by the following will infer the type `Integer` for the variable `r` which might not be what the user intended, giving rise to an error. The type `Integer` can have computational consequences, e.g., using `Integer` division instead of `Real` division in some expression.

```
var r = 3;
```

Another similar example:

```
Real[:] vec = {1,2,3};
```

This is a `Real` vector containing the value {`1.0,2.0,3.0`}. If the type is left out as below, it becomes an `Integer` vector, with possibly unexpected consequences.

```
var vec = {1,2,3};
```

9.2 Types Values of Polymorphic Type Variables in Functions

We have already presented the notion of parameterized `List`, array, and `Option` types. Type variables in MetaModelica can for example appear in function signatures.

For example, the `tuple2GetField1` function takes a tuple of two values having arbitrary types specified by the type variables `Type_a` and `Type_b`, which in the example below will be bound to the types `String` and `Integer`, and returns the first value, e.g.:

```
tuple2GetField1(("x",33)) => "x"
```

The function is parameterized in terms of the types of the first and second fields in the argument tuple, which is apparent from the type signature in its definition:

```
function tuple2GetField1<Type_a,Type_b>
 "Takes a tuple of two values and returns the first value.
  For example,
  tuple2GetField1((true,1)) => true "
  input Tuple<Type_b,Type_b> inTuple;
  output Type_a res;
algorithm
  res := match inTuple
    local Type_a a;
    case (a,_) then a;
  end match;
end tuple2GetField1;
```

Hindley-Milner type inference can is used in this limited context to deduce the actual types of the type variables. By matching the tuple type `Tuple<Type_a,Type_b>` against the actual tuple value `("x",33)`, `Type_a` is inferred to be `String` and `Type_b` to be `Integer`.

9.3 Inferring Types of as-Bound Variables

If an `as`-bound variable in a pattern is declared as a uniontype, the specific type of the variable can be inferred from the pattern it is bound to within the scope of the binding. This is necessary to be able to access field names via dot-notation for an as-bound variable.

In the eval function below this is used to access fields via dot-notation. For example, in the `case x as INTconst(__)` the specific type of x in the scope of the case is inferred to be `Exp.INTconst` whereas in in the `case x as ADDop(__)` the specific type of x in the scope of the case is inferred to be `Exp.ADDop`.

```
function eval
```

```
  input  Exp     inExp;
  output Integer outInteger;
algorithm
  outInteger := match inExp
    local Exp x;
    case x as INTconst(__) then x.INTconst.int;
    case x as ADDop(__) then eval(x.exp1) + eval(x.exp2);
    case x as SUBop(__) then eval(x.exp1) - eval(x.exp2);
    case x as MULop(__) then eval(x.exp1) * eval(x.exp2);
    case x as DIVop(__) then eval(x.exp1) / eval(x.exp2);
    case x as NEGop(__) then –eval(x.exp);
  end match;
end eval;
```

9.4 Inferring Types of Input Variables to Pattern Matching

If an input variable to a pattern matching is declared as a uniontype, the specific type of the variable can be inferred from the pattern it is bound to within the scope of the pattern. This is necessary to be able to access field names via dot-notation for such a variable.

In the eval function below this is used to access fields via dot-notation. For example, within the `case INTconst(__)` the specific type of `inExp` in the scope of the case is inferred to be `Exp.INTconst` whereas in in the `case ADDop(__)` the specific type of `inExp` in the scope of the case is inferred to be `Exp.ADDop`.

```
function eval
  input  Exp     inExp;
  output Integer outInteger;
algorithm
  outInteger := match inExp
    case INTconst(__) then inExp.int;
    case ADDop(__) then eval(inExp.exp1) + eval(inExp.exp2);
    case SUBop(__) then eval(inExp.exp1) - eval(inExp.exp2);
    case MULop(__) then eval(inExp.exp1) * eval(inExp.exp2);
    case DIVop(__) then eval(inExp.exp1) / eval(inExp.exp2);
    case NEGop(__) then –eval(inExp.exp);
  end match;
end eval;
```

Note that the match-expression in general accepts expressions as input for matching, not just variables. However, dot-notation can only be used when a variable is given as input.

9.5 Implicit Type Conversion

Standard Modelica supports implicit type conversion of `Integer` values and variables to `Real` in the following cases:

- Assignments in algorithm sections and declarations, e.g. `realVar := 3` becomes `realVar := 3.0`.
- Equality in equations and declarations, e.g. `realVar = 44` and `55 = realVar` become `realVar = 44.0` and `55.0 = realVar`.
- Mixed `Integer` and `Real` arguments to *overloaded arithmetic operators* are all converted to `Real` if mixed argument types are present, e.g., `realVar + 44` and `55 < realVar` become `realVar + 44.0` and `55.0 < realVar`.
- Mixed `Integer` and `Real` arguments with conversion to Real for *special overloaded operators* like vector construction `{1,1.5,3}` => `{1.0,1.5,3.0}`, matrix construction, if-expressions (`if cond then 1 else 2.5` => `if cond then 1.0 else 2.5`), and ranges (`1: 0.1: 2` => `1.0: 0.1: 2.0`).

Vectorization of functions and operators also makes this applicable to `Integer` and `Real` array arguments, e.g., {1,2,3}+{4.1,5.1,6.1} => {1+4.1,2+5.1,3+6.1} => {1.0+4.1,2.0+5.1,3.0+6.1}.

- Passing `Integer` arguments to `Real` formal parameter at function calls, e.g., `func(44,55)` becomes `func(44.0,55)` if the first formal parameter of `func` has type `Real` and the second type `Integer`.

9.5.1 The { } Constructor and Implicit Conversions to Arrays and Lists

MetaModelica introduces the `List` and `MRArray` types which are subtypes of `Sequence` (Section 6.2). The constructor {} has been generalized to become a `Sequence` constructor that also can be used to create collections of the subtypes `List`, `MRArray`, and `Array`, using implicit conversions depending on the type context. This can be intermixed with implicit conversion of `Integer` elements to `Real`. Conversions of `Sequence` occur in two main kinds of type contexts:

- Assignment conversion of objects of `Sequence` type to a `List`, `MRArray` or `Array` type by assignment, equation equality, or passing of arguments to a formal parameter.
- Conversion of arguments of `Sequence` type to `List`, `MRArray` or `Array` arguments when passed to constructors or operators requiring the same argument type.

For example, `MRarr := {{1,2},{3.5,4.5}} => MRArray(MRArray(1.0,2.0), MRArray(3.5,4.5))` according to conversion of nested sequences to `MRArray`, and conversion of `Integer` to `Real`.

Another example is `arr := {{1.0,2.0},intList} => arr := Array({1.0,2.0},intList)` according to conversion of sequences, `=> arr := Array(List(1.0,2.0),intReal(intList))` since the `Sequence` constructor is converted to `List` and the elements of `intList` to `Real` making all arguments of the constructor have the same type.

If there is any ambiguity, the explicit constructors `List(…)`, `MRArray(…)`, or `Array(…)`can be used instead of {}. See also Section 6.5 for a complete table of the conversion operators between arrays, lists, and MRArrays, and conversions from sequences.

To summarize, the implicit conversions of sequences to lists, arrays, and MRArrays make it possible to conveniently use the curly-brace constructor for construction of these collections in a rather transparent way.

9.5.1.1 Implicit Conversion from Sequence to Array

Sequences containing subsequences of varying lengths, e.g. {{3,4}, {5,6,7}}, cannot be represented as standard Modelica rectangular arrays which require all subarrays to have equal length. An attempt at converting non-rectangular arrays will cause a type error.

In the following note that the `Array` and `array` constructors are completely equivalent, with `array` being the current Modelica 3.2 standard.

The following cases of `Sequence` to `Array` conversion are supported:

- Assignments in algorithm sections and declarations, e.g., `arrVar := {3,4}` becomes `arrVar := array(3,4)`.
- Equality in equations and declarations, e.g. `arrVar = {3,4}` becomes `arrVar = array(3,4)`.
- Passing `Sequence` typed arguments to array formal parameters at function calls, e.g., `func({3,4}, arrVar)` becomes `func(array(3,4), arrVar)` if the formal parameters of `func` have type `Real[:]`.
- Conversion of arguments of `Sequence` type to `Array` when passed to constructors or operators requiring the same argument type. For example, `arr := {{1,2.0},realArr} => arr := Array(Array(1.0,2.0),realArr)`.
- Note: array patterns are not supported in MetaModelica 2.0 but might be in MetaModelica 3.0. Matching an array value (below `arrVar`) against a curly-brace { } sequence constructor pattern.

Here the sequence constructor pattern {…} is converted to the corresponding `Array` constructor pattern, e.g., `match arrVar case {3,_} …` is converted into `match arrVar case array(3,_) …`.

9.5.1.2 Implicit Conversion from Sequence to List

The following cases of implicit conversion of `Sequence` to `List` collections are supported.

- Assignments in algorithm sections and declarations, e.g., `listVar := {3,4}` becomes `listVar := List(3,4)`, and `listVar2 := {{3},{4,5}}` becomes `listVar2 := List(List(3), List(4,5))`.
- Equality in equations and declarations, e.g., e.g., `listVar = {3,4}` becomes `listVar = List(3,4)`, and `listVar2 = {{3},{4,5}}` becomes `listVar2 = List(List(3), List(4,5))`.
- Passing `Sequence` typed arguments to `List` formal parameters at function calls, e.g., `func({{3},{4,5}}, listVar)` becomes `func(List(List(3),List(4,5)), listVar)` if the formal parameters of `func` have types `List< List<Real>>` and `List<Real>` respectively.
- Conversion of arguments of `Sequence` type to `List` when passed to constructors or operators requiring the same argument type. For example, `lst := {{1,2.0},realList}` => `lst := List(List(1.0,2.0),realList)`.
- Matching a list value (below `listVar`) against a curly-brace { } sequence constructor pattern. Here the sequence constructor pattern {…} is converted to the corresponding list constructor pattern, e.g., `match listVar case {3,_} …` is converted into `match listVar case List(3,_) …`

9.5.1.1 Implicit Conversion from Sequence to MRArray

MetaModelica also introduces the `MRArray` globally mutable (ragged) array type described in Section 5.3.2. Collections and patterns of `Sequence` type are converted as follows:

- Assignments in algorithm sections and declarations, e.g., `mrarrayVar := {3,4}` becomes `mrarrayVar := MRArray(3,4)`, and `mrarrayVar2 := {{3},{4,5}}` becomes `mrarrayVar2 := MRArray(MRArray(3),MRArray(4,5))`.
- Equality in equations and declarations, e.g., `mrarrayVar := {3,4}` becomes `mrarrayVar = MRArray(3,4)`, and `mrarrayVar2 = {{3},{4,5}}` becomes `mrarrayVar2 = MRArray( MRArray(3),MRArray(4,5))`.
- Passing list or array arguments to `MRArray` formal parameters at function calls, e.g., `func({{3},{4,5}}, arrayVar)` becomes `func(MRArray(MRArray(3),MRArray(4,5)), arrayMRArray(arrayVar))` if the formal parameters of `func` have types `MRArray< MRArray<Real>>` and `MRArray<Real>` respectively.
- Conversion of arguments of `Sequence` type to `MRArray` when passed to constructors or operators requiring the same argument type. For example, `mrarr := {{1,2.0},realMRArray}` => `mrarr := MRArray(MRArray(1.0,2.0),realMRArray)`.
- Note that `MRArray` patterns are not supported by MetaModelica 2.0 but might be supported by MetaModelica 3.0. We are matching a list value (below `listVar`) against a curly-brace { } sequence constructor pattern. Here the sequence constructor pattern {…} is converted to the corresponding list constructor pattern, e.g., `match listVar case {3,_} …` is converted into `match listVar case List(3,_) …`

Chapter 10

Functions

We have already used MetaModelica functions extensively to express the semantics of a number of small languages, as well as small declarative programs. This chapter gives a more complete presentation of the MetaModelica function construct, its properties, and its usage.

Modelica functions are declarative *mathematical functions*, i.e., a Modelica function always returns the same results given the same argument values. Thus a function call is *referentially transparent*, which means that it keeps the same semantics or meaning independently of from where the function is referenced or called.

The declarative behavior of function calls implies that functions have *no memory* (not being able to store values that can be retrieved in subsequent calls) and *no side effects* (e.g. no update of global variables and no input/output operations). However, it is possible that external functions could have side effects or input/output operations. Moreover, there are built-in functions such as `print` and `tick` with side-effects. See Section 10.6.6 for a discussion of these functions, also see Appendix B.3.9. Both positional and named argument-passing at function calls are supported.

10.1 Function Declaration

The body of a MetaModelica function is a kind of algorithm section that contains procedural algorithmic code to be executed when the function is called. Formal parameters are specified using the `input` keyword, whereas results are denoted using the `output` keyword. This makes the syntax of function definitions quite close to Modelica class definitions.

The structure of a typical function declaration is sketched by the following schematic function example.

```
function functionname <OptTypevar1,OptTypevar2,...>
  input  TypeI1 in1;
  input  TypeI2 in2;
  input  TypeI3 in3  "Comment" annotation(...);
  ...
  output TypeO1 out1;
  output Type02 out2;
  ...
protected
  local variables
  ...
algorithm
  ...
   statements
  ...
end functionname;
```

Comment strings and annotations can be given for any formal parameter declaration, as usual in MetaModelica declarations.

A function can optionally be parameterized in terms of formal type parameters within angle brackets, e.g., as in `<OptTypevar1,OptTypevar2,...>`. It can have optional prefixes, such as `partial`, `impure`, `pure`, `protected`, etc.

All internal parts of a function are optional; i.e., the following is also a legal function:

```
function <functionname>
end <functionname>;
```

The `algorithm` keyword is not needed when the outputs of a function are directly defined by expressions, e.g., as in the following example>

```
function add1  "Add 1 to integer input argument"
  input Integer x;
  output Integer y;
algorithm
  y := x + 1;
end add1;
```

The same example, without an algoritm section:

```
function add1  "Add 1 to integer input argument"
  input Integer x;
  output Integer y := x + 1;
end add1;
```

10.2 Builtin Functions

A number of "standard" builtin primitives are provided by the MetaModelica standard library—in a package called `MetaModelica`. Examples are `intAdd`, `intSub`, `stringAppend`, `listAppend`, etc. A complete list of these primitives can be found in Appendix B.

10.3 Pure and Impure Functions

By default, Modelica functions are *pure*, i.e., are side-effect free with respect to the internal Modelica state (the set of all Modelica variables in a total simulation model), apart from the exceptions mentioned below. This implies the following:

- A *pure* Modelica function is a mathematical function, i.e. calls with the same input arguments always give the same output arguments.
- A *pure* Modelica function is side-effect free with respect to the internal Modelica simulation state. Specifically, the ordering of function calls and the number of calls to a function shall not influence the simulation state.

A Modelica function which does not have the *pure* function properties is *impure* and needs to be declared as stated below.

Note that pure functions enables writing declarative specifications using Modelica. It also makes it possible for Modelica compilers to freely perform algebraic manipulation of expressions containing function calls while still preserving their semantics. For example, a tool may use common subexpression elemination to call a pure function just once, if it is called several times with identical input arguments.

The Modelica translator is responsible for maintaining the pure function property for pure non-external functions. Regarding external functions, the external function implementor is responsible. Note that external functions can have side-effects as long as they do not influence the internal Modelica simulation state, e.g. caching variables for performance or printing trace output to a log file.

With the prefix keyword **`impure`** it is stated that a Modelica function is *impure* and it is only allowed to call such a function from within:

- another function marked with the prefixes `impure` or `pure`

- a when-equation, or
- a when-statement.

Note that a tool is not allowed to perform any optimizations on function calls to an impure function, e.g., reordering calls or common subexpression elimination is not allowed.

It is possible to mark a formal function parameter as `impure`. Only if the function formal parameter is marked `impure`, it is allowed to pass an `impure` function to it. A function having a formal function parameter marked `impure` must be marked `pure` or `impure`.

For example, in a call g(h, …) where h is passed and the formal function parameter delaration inside g is marked `impure` means that g can accept impure functions as argument. Thus, g must be marked `pure` or `impure` by the user. It is an error to not mark the definition of g as `pure` or `impure`.

Furthermore, a function inheriting an impure function must be marked `impure` or `pure`.

With the prefix keyword **`pure`** it is stated that a Modelica function is pure despite the fact that it might be calling an impure function, use external objects, or might be an external function written in a language that allows impure functions.

Below are a few examples:

```
function evaluateLinear  // pure function without the pure keyword
   input  Real a0;
   input  Real a1;
   input  Real x;
   output Real y;
algorithm
   y = a0 + a1 * x;
end evaluateLinear;

impure function receiveRealSignal   // impure function
   input  HardwareDriverID id;
   output Real y;
   external "C" y = receiveSignal(id);
end receiveRealSignal;

// Note: Should add an example with the pure keyword
```

10.4 Failure and Retry in Functions

Two important properties of MetaModelica functions are absent for ordinary Modelica functions:

- Functions in MetaModelica can fail or succeed.
- Retry is supported between cases in a match-expression using `matchcontinue`.

A call to a function can *fail* instead of always returning a result which is the case for standard Modelica functions. This is convenient for the specification writer when expressing semantics, since other possibly matching cases in the function will be applied without needing “try-again” mechanisms to be directly encoded into specifications. The failure handling mechanism can also be used in general declarative programming, e.g. the factorial example previously presented in Section 2.3.2.

This brings us into the topic of case retry. If there is a failure in case, or in one of the functions directly or indirectly called via the local equations of the case, and a match-expression is used, MetaModelica will backtrack (i.e., undo) the part of the “execution” which started from this case, and automatically *continue* with the next case (if there is one) in top-down, left-to-right order. If no case in the function matches and succeeds, then the call to this function will fail. Correct back-tracking is however dependent on avoidance of side-effects in the cases of the specification.

10.5 Argument Passing and Result Values

Any kind of data structure, as well as functions, can be passed as actual arguments in a call to a MetaModelica function. One or more results can be returned from such a call. The issues are discussed in some detail in the following sections.

10.5.1 Tuple Arguments and Results

A Modelica function can have multiple arguments and results. This should not be confused with the case where a MetaModelica tuple type (see Section 7.1) consisting of several constituent types is part of the signature of a function. For example, the function `incrementpair` below accepts a single tuple of two integers and returns a tuple where both integers have been incremented by one..

```
function incrementpair
  input  Tuple<Integer,Integer> inVal;
  output Tuple<Integer,Integer> outVal;
algorithm
  outVal :=
  match inVal
    local Integer x1,x2;
    case (x1,x2) then (x1+1,x2+1);
  end match;
end incrementpair;
```

For example, the call:

```
incrementpair((2,3))
```

gives the result:

```
(3,4)
```

10.5.2 Passing Functions as Arguments to Functions

Functions can be passed as actual parameters at a function call, i.e., as a kind of function parameters. In the example below, the function `add1` is passed as a parameter to the function `map`, which applies its formal parameter `func` to each element of the parameter list.

A function declaration gives rise to both a function type and a single function object with the same name. In this case the function object `add1` is passed as an argument to `map`.

For example, applying the function `add1` to each element in the list `{0,1,2}`, e.g. `map(add1, {0,1,2})`, will give the result list `{1,2,3}`.

```
function add1  "Add 1 to integer input argument"
  input Integer x;
  output Integer y;
algorithm
  y := x+1;
end add1;

function listMap<Type_a,Type_b>
 "Takes a list and a function over the elements of the lists,
  which is applied for each element, producing a new list.
  For example  listMap({1,2,3}, intString) => { "1", "2", "3"}"
  input List<Type_a>  in_aList;
  input FuncType      inFunc;
  output List<Type_b> out_bList;
public
  function FuncType<Type_b>
    input  Type_a in_a;
    output Type_b out_b;
  end FuncType;
algorithm
```

```
  out_bList := match (in_aList,inFunc)
    local
      Type_b first_1;
      List<Type_b> rest_1;
      Type_a first;
      List<Type_a> rest;
      FuncType fn;

    case ({},_) then {};

    case (first :: rest,fn)
      equation
        first_1 = fn(first);
        rest_1 = listMap(rest, fn);
      then first_1 :: rest_1;

  end match;
end listMap;


function main
  ...
  res := listMap({0,1,2}, add1);  /* Pass add1 as a parameter to listMap */
                                  /* In this example res will be {1,2,3}  */
  ...
end main;
```

10.6 Variables and Types in Functions

Except for global constants, MetaModelica variables only occur in functions. Types, including parameterized types, can be explicitly declared in MetaModelica function type signatures.

10.6.1 Type Variables and Parameterized Types in Functions

We have already presented the notion of parameterized list, array, and Option types in Chapter 6. Type variables in MetaModelica can for example appear in function signatures.

For example, the `tuple2GetField1` function takes a tuple of two values having arbitrary types specified by the type variables `Type_a` and `Type_b`, which in the example below will be bound to the types `String` and `Integer`, and returns the first value, e.g.:

```
tuple2GetField1(("x",33)) => "x"
```

The function is parameterized in terms of the types of the first and second fields in the argument tuple, which is apparent from the type signature in its definition:

```
function tuple2GetField1<Type_a,Type_b>
"** Takes a tuple of two values and returns the first value.
 ** For example,
 ** tuple2GetField1((true,1)) => true
 *"
  input Tuple<Type_a,Type_b> inTuple;
  output Type_a res;
algorithm
  res :=
  match (inTuple)
    local Type_a a;
    case (a,_) then a;
  end match;
end tuple2GetField1;
```

10.6.2 Local Variables in Match-Expressions in Functions

Variables in MetaModelica functions consisting of only a match-expression are normally introduced at the beginning of a match-expression. The only exceptions are global constants. There are two kinds of local variables for values:

- *Pattern local variables*, which are given values in patterns to be matched, and declared in a `local` declaration.
- *Ordinary local variables*, which occur on the left hand side of equality signs, e.g.: variable = *expression*, and also need to be declared in a `local` declaration. Result variables can be regarded as a special case of pattern variables, for the trivial pattern consisting of the variable itself.

There are also polymorphic type variables which are introduced in a comma separated list within angle brackets after a function or class name in the corresponding function/class declaration.

- *Polymorphic type variables*, which are declared within angle brackets.

For example, in the function `listThread` below, `Type_a` is a polymorphic type variable for the type of elements in the list, being a subtype of the pre-defined top level type `Any`, `fa`, `rest_a`, `fb`, `rest_b` are pattern variables in the pattern `listThread(fa::rest_a, fb::rest_b)`:

```
function listThread<Type_a>
 "Takes two lists of the same type and threads them together.
  For example,  listThread({1,2,3},{4,5,6}) => {4,1,5,2,6,3}"
  input List<Type_a> inList1;
  input List<Type_a> inList2;
  output List<Type_a> outList;
algorithm
  outList := match (inList1,inList2)
    local
      List<Type_a> rest_a,rest_b;  Type_a fa,fb;
    case ({},{}) then {};
    case (fa :: rest_a, fb :: rest_b)
      then fa :: fb :: listThread(rest_a, rest_b);
  end match;
end listThread;
```

10.6.3 Function Failure Versus Boolean Negation

MetaModelica functions can fail or succeed, whereas conventional functions always succeed in returning some value. See Section 8.1 for a discussion of this topic.

10.6.4 Last Call Optimization – Tail Recursion Removal

A typical problem in declarative programming is the cost of recursion instead of iteration, caused by recursive function calls, where the implementation of each call typically needs a separate allocation of an activation record for local variables, etc. This is costly both in terms of execution time and memory usage.

There is however a special form of declarative recursive formulation called *tail-recursion*. This form allows the compiler to avoid this performance problem by automatically transforming the recursion to an iterative loop that does not need any stack allocation and thereby be as efficient as iteration in imperative programs. This is called the *last call optimization* or *tail-recursion removal*, and is dependent on the following:

- A *tail-recursive* formulation of a function (or function) *calls itself as its last action* before returning.

In the following we give several recursive formulations of the summation function `sum`, both with and without tail-recursion. This function sums integers from `i` to `n` according to the following definition:

```
sum(i,n) = i + (i+1) + ... + (n-1) + n
```

This can be stated as a recursive function:

```
sum(i,n) = if i>n then 0 else i+sum(i+1,n)
```

A recursive MetaModelica function for computing the sum of integers can be expressed as follows:

```
function sum
  input Integer inInteger;
  input Integer in_n;
  output Integer outRes;
algorithm
  outRes :=
  matchcontinue (inInteger,in_n)
    local Integer i,n,i1,res1;
  case (i,n)
    equation
      true = (i>n);  then true;
  case (i,n)
    equation
      false = (i>n);
      i1 = i+1;
      res1 = sum(i1,n); then i+res1;
  end matchcontinue;
end sum;
```

The above function `sum` is *recursive* but not *tail-recursive* since its last action is adding the result `res1` of the `sum` call to `i`, i.e., the recursive call to `sum` is *not* the last action that occurs before returning from the function.

Fortunately, it is possible to reformulate the function into tail-recursive form using the method of accumulating parameters, which we will show in the next section.

Note that when the full MetaModelica language is available, the above `sum` function can be expressed more concisely:

```
function sum
  input Integer i;
  input Integer n;
  output Integer outRes;
algorithm
  outRes := if i>n then 0 else i+sum(i+1,n)
end sum;
```

10.6.5 The Method of Accumulating Parameters for Collecting Results

The method of accumulating parameters is a general method for expressing declarative recursive computations in a way that allows collecting intermediate results during the computation and makes it easier to achieve an efficient tail-recursive formulation.

We reformulate the `sum` function by adding an accumulating input parameter `sumSoFar` to a help function `sumTail`, keeping the counter `i`. When the terminating condition `i>n` occurs the accumulated sum `sumSoFar` is returned. The function `sumTail` is tail-recursive since the call to `sumTail` is the last action that occurs before returning from the function body, i.e.:

```
sum(i,n) = sumTail(i,j,0)

sumTail(i,n,sumSoFar) = if i>n then sumSoFar else sumTail(i+1,n,i+sumSoFar)
```

The functions `sum` and `sumTail` expressed as MetaModelica functions:

```
function sum
  input Integer i;
  input Integer n;
```

```
  output Integer outRes;
algorithm
  outRes := sumTail(i,n,0);
end sum;

function sumTail
  input Integer i;
  input Integer n;
  input Integer sumSoFar;
  output Integer outRes;
algorithm
  outRes := if i>n then sumSoFar else sumTail(i+1,n,i+sumSoFar);
end sumTail;
```

It is easy to see that the function `sumTail` is tail-recursive since the call to `sumTail` is the last computation in the last local equation of the second case.

Another example of a tail-recursive formulation is a revised version of the previous `listThread` function from Section 10.6.2, called `listThreadTail`:

```
listThread(a,b) = listThreadTail(a,b,{})
```

We have introduced an accumulating parameter as the third argument of `listThreadTail`, e.g.:

```
listThreadTail({1,2,3},{4,5,6},{}) => {4,1,5,2,6,3}
```

Its definition follows below:

```
function listThreadTail<Type_a>
 "Takes two lists of the same type and threads them togheter.
  For example,  listThread({1,2,3},{4,5,6}) => {4,1,5,2,6,3}"
  input List<Type_a> inList1;
  input List<Type_a> inList2;
  input List<Type_a> in_accumlst;
  output List<Type_a> outList;
algorithm
  outList := match (inList1,inList2,in_accumlst)
    local
      List<Type_a> rest_a,rest_b,accumlst;  Type_a fa,fb;
    case ({},{},{}) then {};
    case (fa :: rest_a, fb :: rest_b, accumlst)
      then listThreadTail(rest_a, rest_b, fa :: fb :: accumlst);
  end match;
end listThreadTail;
```

10.6.6 Using Side Effects in Specifications

Can side effects such as updating of global data or input/output be used in specifications? Consider the following contrived example:

```
function foo
  input  Real in_x;
  output Real out_y;
algorithm
  out_y := matchcontinue in_x
    local Real x,y;
    case x equation
      print "A"; y = condition_A(x); then y;
    case x equation
      print "A"; y = condition_A(x); then y;
  end matchcontinue;
end foo;
```

The builtin function `print` is called in both cases, giving rise to the side effect of updating the output stream. The intent is that if `condition_A` is fulfilled, `"A"` should be printed and a value returned. On the other hand, if `condition_B` is fulfilled, `"B"` should be printed and some other value returned. The problem occurs if `condition_A` fails. Then backtracking will occur, and the next case (which has the

same matching pattern) will be tried. However, the printing of "A" has already occurred and cannot be undone.

Such problems can be avoided if the code is completely determinate—at most one case in a function matches and backtracking never occurs. Thus we may formulate the following usage rule:

- Only use side-effects in completely deterministic functions for which at most one case matches and backtracking may never occur.

The problem can be avoided by separating the `print` side effect from the locally non-determinate choice, which is put into a side-effect free function `chooseFoo`.

```
function chooseFoo
  input  Real in_x;
  output Real out_y;
algorithm
  out_y := matchcontinue in_x
      local Real x,y;
    case x equation
      y = condition_A(x); then ("A",y);
    case x equation
      y = condition_B(x); then ("B",y);
  end matchcontinue
end chooseFoo;

function foo
  input  Real x;
  output Real y;
protected
  Real z;
algorithm
  (z,y) := chooseFoo(x);
  print(z);
end foo;
```

In the above contrived example, the problem can also be avoided in an even simpler way by just putting `print` after the condition using the fact that the evaluation of the local equations stops after the first local equation that fails:

```
function foo2
  input Real in_x;
  output Real out_y;
algorithm
  out_y := matchcontinue in_x
      local Real x,y;
    case x equation
      y = condition_A(x); print "A"; then y;
    case x equation
      y = condition_B(x); print "B"; then y;
  end matchcontinue;
end foo2;
```

A natural question concerns the circumstances when side effects may occur, since MetaModelica is basically a side-effect free specification language. The following two cases can however give rise to side effects:

- The `print` primitive causes side effects by updating the output stream.
- External C functions which may contain side effects can be called from MetaModelica.

There is also a builtin function `tick`, which generates a new unique (integer) "identifier" at each call—analogous to a random number generator. In order to ensure that each new integer is unique, some global state (e.g. a counter) has to be updated, which is a side effect. However, from the point of view of a semantics specification the actual value from `tick` is irrelevant—only the uniqueness is important. It does not matter if `tick` is called a few extra times and some values are thrown away during

backtracking. Thus, from a practical semantics point of view `tick` may be treated as a side effect free primitive if used in an appropriate way.

10.7 Examples of Higher-Order Programming with Functions

The idea of higher-order functions in declarative/functional programming languages is that functions should be treated as any data object: passed as arguments, assigned to variables, returned as function values, etc.

MetaModelica supports a limited form of higher-order programming: functions can be passed as arguments to other functions, but cannot yet be returned as values or directly assigned as values.

We give three examples of higher-order MetaModelica functions that take another function as a parameter. The functions are the following:

- `listReduce`
- `listMap`
- `listFold`

10.7.1 Reducing a List to a Scalar Using listReduce

The `listReduce` function takes a list and a function argument operating on two elements of the list. The function performs a reduction of the list to a single value using the function passed as an argument.

```
listReduce({1,2,3},intAdd) => 6

function listReduce<TypeA>
  input List<TypeA> inTypeAlist;
  input FuncType inFunc;
  output TypeA outTypeA;
 public
  partial function FuncType
    input TypeA inTypeA1;
    input TypeA inTypeA2;
    output TypeA outTypeA;
  end FuncType;
algorithm
  outTypeA := matchcontinue (inTypeAlist,inFunc)
    local
      TypeA a,b; FuncType r; List<TypeA> xs;
    case (List(a),r) then a;
    case (List(a,b),r) then r(a,b);
    case (a :: b :: (xs as _ :: _),r) then r(r(a,b), listReduce(xs,r));
  end matchcontinue;
end listReduce;
```

10.7.2 Mapping a Function Over a List Using listMap

The `listMap` function takes a list and a function over the elements of the lists, which is applied to each element, producing a new list. For example, `intString` has the signature: (`int => string`)

```
listMap({1,2,3}, intString) => { "1", "2", "3"}

function listMap<TypeA,TypeB>
  input List<TypeA> inVTypeAlist;
  input FuncType inFunc;
  output List<TypeB> out_vTypeBlist;
public
  partial function FuncType
    input TypeA inTypeA;
    output TypeB outTypeB;
```

```
  end FuncType;
algorithm
  out_vTypeBlist := match (inVTypeAlist,inFunc)
    local
      TypeB f1;
      List<TypeB> r1;
      TypeA f;
      List<TypeA> r;
      FuncType fn;

    case ({},_) then {};

    case (f :: r,fn)
      equation
        f1 = fn(f);
        r1 = listMap(r, fn);
      then f1 :: r1;
  end match;
end listMap;
```

The `listFold` function takes a list and a function operating on pairs of a list element and an accumulated value, together with an extra accumulating parameter which is eventually returned as the result value. The third argument is the start value for the accumulating parameter. `listFold` will call the passed function for each element in a sequence, adding to the accumulating parameter value.

```
listFold({1,2,3},intAdd,2) =>  8
```

```
intAdd(1,2) => 3, intAdd(2,3) => 5, intAdd(3,5) => 8
```

```
function listFold<TypeA,TypeB>
  input List<TypeA> inVTypeAlist;
  input FuncType inFunc;
  input TypeB inTypeB;
  output TypeB outTypeB;
 public
  partial function FuncType
    input TypeA inTypeA;
    input TypeB inTypeB;
    output TypeB outTypeB;
  end FuncType;
algorithm
  outTypeB := match (inVTypeAlist,inFunc,inTypeB)
    local
      FuncType r;
      TypeB b,b1,b2;
      TypeA l;
      List<TypeA> lst;

    case ({},r,b) then b;

    case (l :: lst,r,b)
      equation
        b1 = r(l, b);
        b2 = listFold(lst, r, b1);
      then b2;

  end match;
end listFold;
```

Chapter 11

Packages and Import

In Modelica 3.2 an import-clause can occur in one of the following five syntactic forms:

```
import packagename;                                  (qualified import)
import packagename.definitionname;                   (single definition import)
import packagename.*;                                (unqualified import)
import shortpackagename = packagename;               (renaming import)
import shortpackagename = [packagename.]definitionname;   (renaming single def. import)
```

Here *packagename* is the fully qualified name of the imported package including possible dot notation and *definitionname* is the name of an element in a package.

Using **import** *packagename*.* (unqualified import) is convenient, but can cause future maintenance problems if new definitions that clashes with user code are added to the imported package.

However, single definition imports that would avoid that problem are clumsy to write.

11.1 Compact Syntax for Multiple Definition Import

Since multiple single definition imports are clumsy, there is a new compact syntax for multiple definition imports, inspired from the Scala book Section 13.2:

```
import packagename.{defname1,defname2,…};          (multiple definition import)
```

Example:

```
package Pack
 package Expr
   record Foo1 ... end Foo1;
   record Foo2 ... end Foo2;
  ...
 end Expr;
end Pack;
```

Modelica 3.2 single definition imports:

```
import Pack.Expr.Foo1;
import Pack.Expr.Foo2;
...
```

Short multiple definition import:

```
import Pack.Expr.{Foo1,Foo2};
```

This allows you to use:

```
Foo1() and Foo2()
```

11.2 Importing Union Type Record Constructors

Uniontype constructors can be imported using the uniontype name as prefix, similar to the rules for nested models and packages in Modelica.

```
package Pack
 uniontype Expr
   record Foo1 ... end Foo1;
   record Foo2 ... end Foo2;
  ...
 end Expr;
end Pack;
```

You can use `Expr.Foo1()` and `Expr.Foo2()`by using explicit import:

```
import Pack.Expr;
```

This will allow you to use:

```
Expr.foo1() and Expr.foo2()
```

If you import each constructor:

```
import Pack.Expr.*;
```

or

```
import Pack.Expr.{foo1,foo2};
```

or

```
import Pack.Expr.foo1;
import Pack.Expr.foo2;
```

then you can directly use

```
Foo1() and Foo2()
```

11.2.1 Implicit Import of Union Type Record Constructors

Note that in MetaModelica 1.0, and also for the time being in MetaModelica 2.0 (although this will be deprecated), the record constructors of a uniontype are implicitly imported into the scope where the uniontype is declared and are made available in the surrounding package as if they were declared directly in that package.

For the above examples, this will allow you to directly use:

```
Pack.foo1() and Pack.foo2()
```

The constructor names case be used directly in the same scope as the uniontype:

```
Foo1() and Foo2()
```

This feature is currently used in most examples within this book.

Chapter 12

Text Template Language

The MetaModelica language extensions to Modelica described in previous chapters of this book are primarily focused on mechanisms for mapping/transforming models as structured data (AST) into structured data (AST), which is needed in advanced symbolic transformations and compilers.

However, there is an important *subclass* of problems mapping structured data (AST) representations of models into text. Unparsing is one example. Generation of simulation code into C or some other language from a flattened model representation is another example. Yet another use case is model or document generation based on text templates where only (small) parts of the target text needs to be replaced.

The *OpenModelica text template language Susan* is intended to simplify and decrease costs of implementation and maintenance of code generators, unparsers, XML emitters, etc. from intermediate code (abstract syntax tree – AST, lower level tree IR, all in MetaModelica) to text. The generated code can for example be in C, C#, Java, XML, or some other language.

The text template language is a domain specific language for text template based programming. It is *different* from but strongly related to MetaModelica, and is compiled into MetaModelica.

We believe that providing a text template language for Modelica may fulfill a need for an easier-to-use approach to a class of applications in model transformation based on conversion of structure into text. Particularly, this is useful to implement multiple code generators, i.e., regargeting the OpenModelica compiler simply by specifying a package of templates for the new target language.

12.1 Definition of Text Template Language

In this section we try to be more precise regarding what is meant by the notion of text template language, template and template function.

Definition 1. Template Language. A template language is a language for specifying the transformation of structured data into a textual target data representation, by the use of a parameterized object "the template" and constructs for specifying the template and the passing of actual parameters into the template.

One could generalize the notion of template language to cover target language representations that are not textual. However, in the following we only concern ourselves with textual template languages.

Definition 2. Template and Template Function. A *template function* is a function from a set of attributes/parameters to a textual data structure.

A *template* is a text string with holes in it. The holes are filled by evaluating expressions that are converted to text when evaluating the template body. More formally, we can use the definition from (Parr 2006) slightly adapted:

A template function is a function that maps a set of attributes to a textual data structure. It can be specified via an alternating list of text strings, t_i, and expressions, e_i, that are functions of attributes a_i:

$F(a_1, a_2, ..., a_m) ::= t_0\, e_0 ... t_i\, e_i\, t_{i+1} ... t_n\, e_n\, t_{n+1}$

where t_i may be the empty string and e_i is restricted computationally and syntactically to enforce strict model-view separation, see Section 12.2 and (Parr 2009). The e_i are distinguished from the surrounding text strings by bracket symbols. Some design alternatives are angle brackets `<...>`, dollar sign `$...$`, or combined `<%...%>` as in our Text template language. Evaluating a template involves traversing and concatenating all t_i and e_i expression results.

Definition 3. Textual Data Structure. A textual data structure has text data such as strings of characters as leaf elements. Examples of textual data are: a string, a list (or nested list structure) of strings, an array of strings, or a text file containing a single (large) string. A textual data structure should efficiently be able to convert (flattened) into a string or text file. In the Modelica template language Susan, the textual data structure is the predefined `Text` type.

12.2 Design Principles

The current design and syntax is influenced by other text template languages, especially the Stringtemplate language, as well as by languages such as Modelica, MetaModelica, C, C#, and F#.

The template language has been designed to be strongly typed and efficient. It is compiled into MetaModelica and not interpreted as many other text template languages. This makes it very efficient.

Similar to Stringtemplate, is designed to follow the model-view-controller concept, and to be a simple functional-style language.

- model – the intermediate tree (AST), to be converted to text according to the view.
- view – the mapping to text provided by the template functions.
- controller – the (sometimes conditional) traversal of the tree to create a view from the model.

The value of this principle is strongly argued in (Parr 2004, according to experience with the ST functional template language (Parr 2006) in the StringTemplate system. Such separation gives more flexibility (multiple views), easier maintainability, better reuse, more ease-of-use, etc.

It is argued that the template language should be kept simple, program computation logic should not be too much intertwined with emitting text. If complex computation needs to be done, it should instead be done on the model (in our case the AST).

Therefore, the Text template language language has been designed be simple, only provide a *mapping to text*, *simple conditional tests*, and *pattern matching*. If more complicated computations should be done, it should be done on the model data structure (the tree), before transferring it to the template text output phase.

There are many template languages intended for C and Java users, but this is currently the only text template language adapted for Modelica and MetaModelica users.

The text template language is strongly typed and compiled into efficient MetaModelica code. For example, the current OpenModelica C code generator written in the template language is faster than the previous handwritten C code generator.

12.3 Preliminaries

12.3.1 Predefined Text Data Type

The template language supports a predefined `Text` data type. The implicit result type of all template functions is type `Text`. Values of type `Text` can be very efficiently converted to `String`, or may sometimes be `String`. Buffers passed as reference parameters are always of type `Text`.

12.3.2 Differences and Similarities Between the Text Template Language and MetaModelica

As previously stated, the text template language Susan is different from but related to MetaModelica, and compiles into MetaModelica. It has several domain specific language elements. Furthermore, its design is strongly influenced by functional language constructs in the Stringtemplate and F# languages. The following briefly summarizes some differences and similarities compared to MetaModelica:

- Text template functions are started by the keyword `template`.
- The text template function output variable is not specified. It is always of type `Text`.
- The text template function input formal parameters are declared in the C and Java style syntax of a parenthesized comma-separated list of type-name parameter-name pairs, rather than using the `input` keyword as in Modelica and MetaModelica.
- A local variable is introduced by a `let`-binding and is bound throughout the rest of the expression.
- There is no need for an ending semicolon after a let-binding declaration.
- The match-expression lacks an optional local equation or algorithm section.
- The cases in a match-expression has optional fall-through as in a C switch, i.e., there might be several case-patterns prefixing return of a result, rather than requiring a separate end of branch for each case as in a Java switch or MetaModelica match.
- After a pattern involving a constructor in match case or in an iterator expression, the field names of the corresponding record constructor become implicitly available in the local scope (Section 12.7.5), as compared to MetaModelica, where dot-notation needs to be used (Section 7.2.5).
- Instead of the array/list-comprehension construct used in MetaModelica (Section 6.6), which would have appeared as `{template-expression for elem in element-list}`, Susan uses an arrow-syntax trinary operator: `element-list |> elem => template-expression`, (Section 12.8).

12.4 Template Text-with-hole Constructors and Template Holes

A text template is a text with holes. Inside a hole there can be any valid template expression. For example, the following text has three holes. Inside a hole is is possible to have a general template expression, of which the most simple form is just a name. Such a template expression is evaluated and converted to text during the evaluation of the template.

```
This is text in the template. This is text in the template. This is text in the
template. <%Hole-templ-expr1%> This is text in the template. This is text in the
template. This is text in the template. This <%Hole-templ-expr2%>is text in the
<%Hole-templ-expr3%>template. This is text in the template.  This is text in the
template. This is text in the template. This is text in the template...
```

Template holes are started by <% and ended by %>, i.e., as in <%name%>. When <% is needed as text, its first character need to be ecaped by \, i.e., \<%. There are currently two forms of template text-with hole constructors:

- Single-quote ' ' text-with-hole constructor. All characters are included verbatim as-is, except holes starting with <% and ' which ends the text. Those can be included by prefixing with the escape code backslash as in \<% and \'.
- Multi-Line << >> text-with-hole constructor. The template text starts on the line after << and ends including the line before >>. All characters are included verbatim as-is, except holes starting with <% and >> which ends the text. Those can be included by prefixing with the escape code backslash as in \<% and \<<.

It is actually possible to have multiple lines also in the single-quote variant, but then line-counting, alignment and indentation options do not work. Such options are only supported by the Multi-line variant.

Example of the single-quote ' ' text-with-hole constructor:

```
'Output text <%templ-expr%>.'
```

Example of the Multi-Line << >> text-with-hole constructor:

```
<<
Output text <%templ-expr%>.
>>
```

Example within a template function

```
template functionInput(ModelInfo modelInfo) ::=
match modelInfo
case MODELINFO(vars=SIMVARS) then
<<
int input_function()
{
  <% (vars.inputVars |> SIMVAR(__) hasindex myindex0 =>
    '<%cref(name)%> = localData->inputVars[<%index0%>];') ;separator="\n"%>
  return 0;
}
>>
end match
end functionInput;
```

12.5 Template Expressions

Template expressions can consist of the following:

- Simple names, see Section 12.5.2.
- Quoted names, see Section 12.5.2.
- Double-quoted string constants, see Section 12.5.3.
- Single-quote text-with-hole constructors, see Section 12.4.
- Multi-line text-with-hole constructors, see Section 12.4.
- Conditional expressions, see Section 12.5.6.
- Match-expressions, see Section 12.7.
- Let-expressions, see Section 12.9.
- Function calls, see Section 12.6.3.
- Iterator expressions, see Section 12.8.
- Parenthesized expressions, see Section 12.5.4.
- List-constructors, see Section 12.5.7.
- Option expressions, see Section 12.10.

12.5.1 Automatic Conversion to String Data

Data retrieved from bound variables or returned from called MetaModelica functions in template expressions is automatically converted to string values according to the following.

Any auto-to-string-convertible bound value can be used.

- Automatic to-string conversion applies to the elementary types: `String`, `Integer`, `Real`, `Boolean`.

- Values of `Option` type are output for `SOME` values when the option type is auto-to-string-convertible (recursively).
- values of type `list` and `Array` are concatenated when element type is auto-to-string-convertible.

12.5.2 Bound Variable Reference, name or $'name'

A variable bound to a value is referenced by using its name, e.g. `valueName`, within a template expression (but not in the actual verbatim template text). Syntax:

```
valueName
```

An alternative dollar-quoted variant is similar to but different from Modelica's single-quoted identifiers:

```
$'valueName'
```

The `$'valueName'` means the same identifier as `valueName`, whereas the Modelica `'valueName'` includes the single-quotes in the identifier.

The value of a referenced bound variable name is retrieved and automatically converted to text according to Section 12.5.1. Any auto-to-string-convertible bound value can be used..

Examples where dollar sign or space or + is in the identifier, or just an ordinary name:

```
$'quoted id+23121'
$'$'
ordinaryName
```

Examples when the keyword `let` needs to be an identifier, e.g. as in these examples:

```
field.$'let'
case RECORD($'let'=FOO) then ...
```

12.5.3 Verbatim String Constants

Double-quoted string constants, e.g. `"string constant"`, are available and exactly respect Modelica stirng semantics, which is defined as follows:

```
STRING = """ { S-CHAR | S-ESCAPE } """
S-CHAR = any member of the Unicode character set (http://www.unicode.org; but use UTF-8
         for storing on files) except double-quote """, and backslash "\"
```

Escape codes for certain control characters can be given in strings as follows and are defined in the same way as the C99 standard:

```
S-ESCAPE = "\'" | "\"" | "\?" | "\\" |
           "\a" | "\b" | "\f" | "\n" | "\r" | "\t" | "\v"
```

12.5.4 Parentheses

Parentheses are allowed, and has the normal interpretation used in almost all programming langaugs, i.e. giving priority in the order of evaluation. Expressions in innermost parenthesis will be evaluated first.

12.5.5 Reduction Expressions

If reduction to text is desired, you should use: '`<< expression-to-reduce >>`', which will reduce it to a single text item. This is usually not needed, since eventually the whole template expression including its parts will be reduced to text before being output.

12.5.6 Conditional Expressions

The conditional expression evaluates *templ-exp*$_1$ as the result of the expression when the condition is satisfied, otherwise value of *templ-exp*$_2$ is the result. When the **else** branch is not specified, an empty string is implied.

Syntax:

if [**not**]$_{\text{opt}}$ *condition* **then** *templ-exp*$_1$
[**else** *templ-exp*$_2$]$_{\text{opt}}$

The `condition` is intended to test values for their non-zero/zero-like values. Values of these types are allowed with the following semantics. Zero-like values are treated as false in the Boolean sense:

```
Boolean                 true       / false
Integer or Real         non-0      / 0
String                  non-empty  / ""
list or Array           non-empty  / { } empty list/array
Option                  SOME       / NONE
```

However, testing for the result of a template function (returning result of type `Text`) is not allowed:

```
if templ() then ...    // Error, cannot test conditionally on template function!
```

Example:

```
template globalDataVarNamesArray(String name, list<SimVar> items) ::=
if items then
<<
char* <%name%>[<%listLength(items)%>] = {<% (items |> SIMVAR(__) =>
  '"<%crefSubscript(origName)%>"' ) ;separator=", "%>};
>>
else
<<
char* <%name%>[1] = {""};
>>;
end globalDataVarNamesArray;
```

12.5.7 List Constructor {}

A list constructor {} is useful in conjunction with conditional insertion of separators.

The separator is only inserted if there are two or more non-empty expressions (i.e., expressions not resulting in empty strings).

```
{ expr1, expr2 } ;separator = ","
```

This can be used for construction of a general list with many elements, as in the following:

```
{ expr1, expr2, expr3, ... exprN } ;separator = ","
```

Example using {}:

```
<<
char var_attr[NX+NY+NP] = {
    <% { (vars.stateVars |> SIMVAR(__) =>
        '<%globalDataAttrInt(type_)%>+<%globalDataDiscAttrInt(isDiscrete)%> /*
<%cref(origName)%> */'
      ",\n"),
      (vars.algVars of SIMVAR(__) =>
        '<globalDataAttrInt(type_)%>+<%globalDataDiscAttrInt(isDiscrete)%> /*
<%cref(origName)%> */'
      ",\n"),
      (vars.paramVars of SIMVAR(__)=>
        '<%globalDataAttrInt(type_)%>+<%globalDataDiscAttrInt(isDiscrete)%> /*
<%cref(origName)%> */'
       ",\n") }
```

```
    ;separator=",\n"%>
```

12.6 Template Function Declaration and Call

12.6.1 Template Function Declaration

A template function is declared as follows:

```
template funcname(Argtype1 arg1, Argtype2, arg2, ...) "Optcomment" ::=
templatebody
end funcname;
```

The implicit result type of the template function is `Text`, which always holds and need not be specified.

Example:

```
template functionInput(ModelInfo modelInfo) ::=
  match modelInfo
  case MODELINFO(vars=SIMVARS) then
<<
int input_function()
{
  <% (vars.inputVars |> SIMVAR(__) hasindex index0 =>
    '<%cref(name)%> = localData->inputVars[<%index0%>];') ;separator="\n"%>
  return 0;
}
>>
  end match
end functionInput;
```

12.6.2 Declaring External Imported MetaModelica Functions

A MetaModelica function that is to be called from within a template expression may have at most one output result and needs to be declared in an interface package, see Section 12.11.

The signature of the function is specified in MetaModelica syntax for the package it belongs to.

For example, the MetaModelica function `crefSubIsScalar` below is specified as an external function that can be imported from package `SimCode`.

```
interface package MyInterface

package SimCode

  function crefSubIsScalar
    input DAE.ComponentRef cref;
    output Boolean isScalar;
  end crefSubIsScalar;

end SimCode;

end MyInterface;
```

12.6.3 Template Function Call

Template functions are called in the same way as functions in general, e.g.:

```
fname(arg1, arg2, ... argN);
```

Buffer arguments passed to reference formal parameters need to be prefixed by & in the call. The `Text` result of the template evaluated with the actual parameters is the output of the template function call.

Formal parameters are strongly typed. Automatic to-string conversion of actual parameters applies when appropriate.

12.6.4 Call of Imported External MetaModelica or C Functions

It is possible to call external MetaModelica functions from template functions. Such external functions need to be declared (Section 12.6.2) in an interface package (Section 12.11) which must be imported through an import statement into the package where the external functions are called.

You can use the return value of an external function natively with its original type, have the return value automatically converted to string, or call functions that return no value. Only external functions which return zero or one arguments can be used. Examples of these three cases follows.

12.6.4.1 Using return value natively with its original type

The return value from a called function can be used as is without being converted to text. For example:

```
template crefSubIsScalarHuman(DAE.ComponentRef cref) ::=
  if crefSubIsScalar(cref) then
    "this cref has scalar subscript"
  else
    "this cref does not have scalar subscript"
end crefSubIsScalarHuman;
```

Here the return value from the external imported function `crefSubIsScalar` is a boolean which is used as a boolean in the if-expression. The return value could also have been stored in a variable `resBool` as follows:

```
template crefSubIsScalarHuman(DAE.ComponentRef cref) ::=
  let resBool = crefSubIsScalar(cref)
  if resBool then
    "this cref has scalar subscript"
  else
    "this cref does not have scalar subscript"
end crefSubIsScalarHuman;
```

Alternatively the return value from `crefSubIsScalar` could be passed immediately to another template function `crefSubIsScalarHumanFromBool` like this:

```
template crefSubIsScalarHuman(DAE.ComponentRef cref) ::=
  crefSubIsScalarHumanFromBool(crefSubIsScalar(cref))
end crefSubIsScalarHuman;

template crefSubIsScalarHumanFromBool(Boolean resBool) ::=
  if resBool then
    "this cref has scalar subscript"
  else
    "this cref does not have scalar subscript"
end crefSubIsScalarHumanFromBool;
```

The return value could be used natively in other contexts as well where it makes sense.

12.6.4.2 Call with return value converted to string

The `tmpTick` external function called below returns an integer that is automatically converted to a string when used in a template expression.

```
template uniqueName() ::=
  let uniqName = 'tmp<%System.tmpTick()&>'
  uniqName
end uniqueName;
```

Or just like the following:

```
template uniqueName() ::=
```

```
  'tmp<%System.tmpTick()%>'
end uniqueName;
```

12.6.4.3 Call with empty return value

It is possible to call an external function which has no return value within the following variant of let-expression:

```
let () = noFuncRet() body-expr
```

For example,

```
template writeSomeTextToFile() ::=
  let content = "some text"
  let () = textFile(content, "file_name.txt")  // Call with empty result, e.g. to create file.
  () // This template function returns an empty result, e.g. as void in C, only its side effect is used.
end writeSomeTextToFile;
```

12.6.5 Declaring Reference (buffer) Formal Parameters in a Template Function

In a template function header, & is used before the reference parameter name, and the type in such cases is always `Text`.

Example:

```
template funcname(Argtype1 arg1, Text &arg2Reference) ::= ...
```

12.6.6 Using Reference (buffer) Formal Parameters in a Template Function

A reference parameter formal parameter always must be prefixed by & when it is used, e.g. when when passing it to another function or updating it in a `let ... +=` text append statement (Section 12.9.3).

```
& buffername
```

In the template function `functionBody` we declare a buffer `varDecls` that can be updated. This buffer is passed to the template function `tempDecl`, prefixed with & in the call.

Further down, within the template function `tempDecl`, a side effect is performed on the reference parameter `varDecls`, it is updated, i.e., appended to by the `&varDecls +=` operation.

```
template functionBody(Function fn) ::=
  match fn
   case FUNCTION(__) then
     ...
     let &varDecls = buffer ""
     let retVar = tempDecl("modelica_real", &varDecls)  // inserted & here
     ...
     <<
     ..
     {
       ...
       <%varDecls%>
     }
     >>
  end match
end functionBody;

template tempDecl(String ty, Text &varDecls) ::=
  let newVar = 'tmp<%System.tmpTick()%>'
  let &varDecls += '<%ty%> <%newVar%>; <%\n%>'   // varDecls is updated,
appended
  newVar  // return newVar
end tempDecl;
```

12.6.7 Allowed Side-Effects in Template Functions

The only direct side-effects allowed in template functions are append (+=) operations on buffer variables. Side effects can also occur through calls to external functions.

12.7 Match Expressions and the Idea of Pattern Matching

The match expression in the template language. is mostly used for discrimination of tree structure nodes (abstract syntax) of some MetaModelica uniontype. First we give a brief introduction to union types and the idea of pattern matching.

12.7.1 The MetaModelica uniontype Construct

To be able to declare the type of abstract syntax trees we introduce the `uniontype` construct.

- A union type specifies a union of one or more record types.
- Union types can be recursive – they can reference themselves.

A common usage is to specify the types of abstract syntax trees. In this particular case the following holds for the `Exp` union type:

- The `Exp` type is a union type of six record types
- Its record constructors are `INTConst`, `ADDop`, `SUBop`, `MULop`, `DIVop`, and `NEGop`.

The `Exp` union type is declared below. Its constructors are used to build nodes of the abstract syntax trees for the Exp language.

```
uniontype Exp
  record  INTconst  Integer int;       end INTconst;
  record  ADDop  Exp exp1;  Exp exp2;  end ADDop;
  record  SUBop  Exp exp1;  Exp exp2;  end SUBop;
  record  MULop  Exp exp1;  Exp exp2;  end MULop;
  record  DIVop  Exp exp1;  Exp exp2;  end DIVop;
  record  NEGop  Exp exp;              end NEGop;
end Exp;
```

Using the `Exp` abstract syntax definition, the abstract syntax tree representation of the simple expression 12 + 5 * 13 will be as shown in Figure 2-1. The `Integer` data type is predefined. Other predefined data types are `Real`, `Boolean`, and `String` as well as the parametric type constructors `array`, `list`, `tuple`, and `Option`.

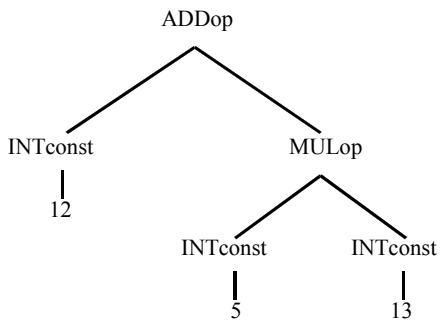

Figure 12-1. Abstract syntax tree of 12 + 5 * 13 in the language Exp.

The `uniontype` declaration defines a union type `Exp` and constructors (in the figure: `ADDop`, `MULop`, `INTconst`) for each node type in the abstract syntax tree, as well as the types of the child nodes.

All union type referenced in the template functions need to be declared in an interface package, see Section 12.11.

1.1.1 Match Expressions and Pattern Matchin3

The match expression is mostly used for discriminating of tree structure nodes of union types. The syntax is as follows, where an optional `else` can appear last in the list of cases.

```
match value-exp
  case pattern-exp1 then templ-exp1
  case pattern-exp2 then templ-exp2
  ...
  [else pattern-expn then templ-expn]opt
[end match]opt
```

The `templ-exp`$_i$ template expression of the first case that matches the `value-exp` against its `pattern-exp`$_i$ is evaluated as the result of the whole expression.

- The value computed by the expression after the `match` keyword is matched against each of the patterns after the `case` keywords in order; if one match fails the next is tried until there are no more case-branches in which case (if present) the else-branch is executed.
- When no pattern matches, the result is empty string.
- When a pattern matches, all pattern variables in the pattern become bound to corresponding parts in the structured value `value-exp`. Also, the scope of the top-most record constructor, if present in the pattern, is opened to make all its fields available as read-only variables, see Section 12.7.5.

12.7.1.1 Simple Pattern Matching

A very simple example of a match-expression is the following code fragment, which returns a number corresponding to a given input string. The pattern matching is very simple – just compare the string value of `s` with one of the constant pattern strings `"one"`, `"two"` or `"three"`, and if none of these matches return 0 since the wildcard pattern `_` (underscore) matches anything.

```
match s
  case "one"   then 1
  case "two"   then 2
  case "three" then 3
  case _       then 0
end match
```

Using an else-branch we could instead have written:

```
match s
  case "one"   then 1
  case "two"   then 2
  case "three" then 3
  else   0
end match
```

The following is an example of pattern matching against a tree structure as in Figure 2-1, e.g.

```
match inExp
  case INTconst(int=v1) then ...
  case ADDop(exp1=e1,exp2=e2) then ...
  case SUBop(__) then ...
  case MULop(__) then ...
  case DIVop(__) then ...
  case NEGop(__) then ...
end match
```

The first case uses named pattern pattern matching (Section 12.7.3 where the single named field of the `INTconst` record is `int`, and the pattern variable is `v1`:

```
case INTconst(int=v1) then ...
```

The interpretation is to match the `inExp` value against the special case pattern `INTconst(int=v1)` If there is a match, the pattern variable `v1` will be bound to the corresponding part of the tree. For the left subtree in Figure 2-1, `INTCONST(12)`, `v1` will be bound to 12.

We now turn to the second rule, which has a named pattern for the `ADDop` node:

```
case ADDop(exp1=e1,exp2=e2) then ...
```

For this case to apply, the pattern `ADDop(exp1=e1,exp2=e2)` must match the actual `inExp` value, which is an abstract syntax tree. If there is a match, the variables `e1` and `e2` will be bound the two child nodes of the `ADDop` node, respectively, visible in Figure 2-1.

A more convenient way of matching is to use the implicit scope opening (Section 12.7.5) of patterns that match. Instead of specifying a named pattern with a number of field names and pattern variables, one can use the following form that matches any `ADDop` node:

```
case ADDop(__) then ...
```

The scope of the `ADDop` node is automatically opened (Section 12.7.5), to make the fields `exp1` and `exp2` available in the current scope bound to the corresponding subtrees of the `inExp` value. Thus, it is unnecessary to introduce the pattern variables `e1` and `e2`. Instead we can use `exp1` and `exp2` directly to refer to the subtrees of the `ADDop` node.

12.7.1.2 Using Pattern Matching in Template Functions

In the following example the template function `exp` is calling itself recursively. The scope of the `ICONST` constructor is automatically opened to make its field called `value` available, and the `PLUS` constructor is opened automatically by using the `PLUS(__)` pattern, see Section 12.7.3 12.7.5.

```
template exp(Exp ep) ::=
  match ep
  case ICONST(__) then value
  case PLUS(__)   then '(<%exp(lhs)%> + <%exp(rhs)%>)'  // Open scope, see Section
12.7.5
end exp;
```

where the type `Exp` is:

```
uniontype Exp
  record ICONST
    Integer value;
  end ICONST;
  record PLUS
    Exp lhs; Exp rhs;
  end PLUS;
end Exp;
```

The `end match` is usually optional, but mandatory in case of nested match expressions as below:

```
match a
 case RECORD1(__) then
  match b
   case RECORD2(__) then expr
  end match
 case ...
```

The above template function exp with an optional `end match` added:

```
template exp(Exp ep) ::=
  match ep
    case ICONST(__) then value
    case PLUS(__)   then '(<%exp(lhs)%> + <%exp(rhs)%>)'  // Open scope, see Section
12.7.5
  end match
end exp;
```

12.7.2 Pattern Expressions in General

Forms of patterns allowed:

- Constant value, e.g. `1`, `"foo"`, `3.14`, etc.
- Constructor with parenthesis and double underscore, e.g. `RELATION(__)` in the example below, with implicit opened record scope, see Section 12.7.5.
- Constructor with parenthesis and named pattern variables, `CONSTRUCTOR(name1=pat1, name2=pat2, ...)` e.g. the `CALL(....)` pattern in the example below.
- A tuple constructor `(arg1, arg2, arg3, ...)` using parentheses.
- A list constructor `{arg1, arg2, arg3, ...}` using curly braces.
- An underscore _ which matches anything.
- A single identifier, `name`, acts as a pattern that can be bound to anything.
- An `as`-expression, e.g. `name` **`as`** `pattern-expr`, Section 12.7.4.

Example:

```
template zeroCrossingTpl(Integer index, Exp relation, Text &varDecls) ::=
match relation
case RELATION(__) then
  let preExp = ""
  let e1 = daeExp(exp1, contextOther, preExp, &varDecls)
  let op = zeroCrossingOpFunc(operator)
  let e2 = daeExp(exp2, contextOther, preExp, &varDecls)
  <<
  <%preExp%>
  ZEROCROSSING(<%index%>, <%op%>(<%e1%>, <%e2%>));
  >>;
case CALL(path=IDENT(name="sample"), expLst={start, interval}) then
  let preExp = ""
  let e1 = daeExp(start, contextOther, preExp, &varDecls)
  let e2 = daeExp(interval, contextOther, preExp, &varDecls)
  <<
  <%preExp%>
  ZEROCROSSING(<%index%>, Sample(*t, <%e1%>, <%e2%>));
  >>;
case _ then
  <<
  ZERO CROSSING ERROR
  >>
end zeroCrossingTpl;
```

12.7.3 Record Constructor Pattern Expressions

The following two forms are possible:

- Constructor with parenthesis and double underscore, e.g. `RELATION(__)` in the example below, with implicit opened record scope, see Section 12.7.5.
- Constructor with parenthesis and named pattern variables, `CONSTRUCTOR(name1=pat1, name2=pat2, ...)`.

Some forms of patterns:

```
REC(field = ASUB())   // constructor with no fields
REC(field = ASUB(field1=_))  // constructor with one field named field1
REC(field = ASUB(__))  // constructor with any number of fields
REC(field = ASUBB)  (mis-spelling of ASUB, then becomes a pattern variable ASUBB
```

12.7.4 Pattern Expression Variable Binding Using as

The following pattern matches `pattern-expression` and binds the variable `var` to the matched value if the match is successful

```
var as pattern-expression
```

This construct is the same in MetaModelica and Text template language, and essentially the same as what is found in several functional language.:

12.7.5 Implicit Opening of Record Constructor Scopes in Patterns

A record constructor pattern in a match-expression case-rule opens the scope of the record and make all the record fields available (read-only). (This is similar to instanceof in Java)

For example in the `ASSIGN(__)` pattern below, the record fields `lhs` and `rhs` becomes available. In the `WHILE(__)` pattern below, the condition and statements fields become available.

Only the scope of the outermost constructor in a nested constructor pattern is opened.

Example:

```
template statement(Statement stmt) ::=
  match stmt
  case ASSIGN(__) then <<
<%exp(lhs)%> = <%exp(rhs)%>;
  >>
  case WHILE(__)  then <<
while(<%exp(condition)%>) {      // call to template function exp
  <%statements |> st => statement(st) ;separator="\n"%>
}
  >>;
end statement;
```

for

```
uniontype Statement
  record ASSIGN
    Exp lhs; Exp rhs;
  end ASSIGN;
  record WHILE
    Exp condition;
    list<Statement> statements;
  end WHILE;
end Statement;
```

Only the scope for the outermost constructor `REC1`, is opened in a nested pattern expression. Use `as`-notation for the nested ones.

Example, with both `REC1` and `REC2` containing `field1`:

```
REC1(...  x as REC2(...) ...)
```

Use of `field1` will denote `field1` within `REC1`, and `x.field1` will denote `field1` within `REC2`.

12.8 Iterator Expressions

Iterator expressions behave similarly to array or list comprehensions in functional languages, but has the arguments in a different order. The following simple variant is a trinary operator that iterates elem over the elements in the element-list and constructs a new list from the template-expressions.

```
element-list |> elem => template-expression
```

This is equivalent to a Modelica iterator expression (also called array- or list-comprehension):

```
{template-expression for elem in element-list}
```

For example, the following iterator expression:

```
{"a", "b", "c"} |> x => 'U<%x%>!')
```

would produce the following list of items:

```
{"Ua!", "Ub!", "Uc!"}
```

that is eventually reduced and concatenated to a single text string:

```
"Ua!Ub!Uc!"
```

The general form of the operator allows a general pattern to iterate over and match the elements in `element-list`. Only the elements that match `elem-pattrn` will be forwarded and used to construct instances of `template-expression`. This is typically used for filtering applications.

```
element-list |> elem-pattrn => template-expression
```

The operator has the following properties:

- It is a trinary operator with three operands. It also has an optional form as a quad operator with four operands when the `hasindex` keyword is present, see Section 12.8.1.
- It has an expression as its third argument, not a function as with the related map function.
- Being a non-associative operator forces the sues parentheses for more clear readability since the evaluation order is always visible from the syntax.
- The left-most argument `elements` must be a list or an array.
- The left-to-right property of the operator can be used to write a series of such operations in a left-to-right fashion – the results of an iterator expression to the left can immediately be fed into an iterator expression to the right – sometimes known as piping.

The iterator expression is often used together with a separator and other options applicable for multi-result values like lists.

Example 1.

```
template gentlemen(list<String> names) ::= <<
Hello <%(names |> name => 'Mr. <%name%>') ;separator=", "%>!
>>
end gentlemen;
```

The output of the template function call `gentlemen({Adam, Eric, Carl})` will be:

```
Hello Mr Adam, Mr Eric, Mr Carl!
```

```
template pairList(list<tuple<String,Integer>> pairs) ::=
<<
Pairs: <%pairs |> (s,i) => '(<%i%>,<%s%>)' \n ;anchor%>.
>>
end pairList;
```

This filters out only values of the record type `ICONST`, see also the match expression Section 12.7.

```
template intConstantsList(list<Exp> expLst) ::=
  (expLst |> ICONST(__) => value ;separator=", ")
end intConstantsList;
```

An example where a list of variable declarations is generated:

```
<%variables |> var as VARIABLE(__) => '<%varType(var)%> <%cref(var.name)%>;'
 ;separator="\n"%>
```

Example:

```
let removedPart = (removedEquations |>  eq  =>
  '<%equation_(eq, contextSimulationNonDiscrete, varDecls)%>' ;separator="\n")
```

Example:

```
(list1 |> x
  => 'I love <%x%> do') |> y   => 'Dumb <%y%>;'
```

More examples:

Ex 1:

```
<%zeroCrossingsNeedSave |> vars => (
  <<
  case <%vars.index0%>:
    <%vars |> SIMVAR(__)=>'save(<%cref(name)%>);' ;separator="\n"%>
    break;
  >> )
;separator ="\n"%>
```

Ex 2: The `includes` binding is from opening the `EXTERNAL_FUNCTION` constructor scope.

```
  <%functions |> EXTERNAL_FUNCTION(__)=>
    (includes ;separator= "\n") ;separator="\n"%>
```

12.8.1 Iterator Expressions with Iteration Index Values

There is a quad operand version of the iterator expression operator with the optional **`hasindex`** keyword (and the additionally optional `from` keyword), followed by an identifier, i.e., a total of four operands:

```
elements |> elem-pattrn [hasindex myindex0 [fromindex startindex ] ] =>
template-expression
```

The variable after `hasindex` gives the ordinal number of the corresponding element in elements starting counting from 0 as default, but it is possible to start from another number, e.g., 1, by using `fromindex 1`.

The following examples use the `hasindex` keyword. One example is using the optional `fromindex` keyword to specify the start index offset to specify non-zero start indexes such as having 1 as the lowest index (as in Modelica):

```
<multi-val-expr |> el hasindex myindex0 => templ(el, myindex0) ;options>
<multi-val-expr |> el hasindex myindex1 fromindex 1 => templ(el, myindex1)>
```

12.9 Let Expressions with Name Bindings and Text Buffers

12.9.1 let Binding of Local Named Text Values

The language allows local definitions that can be referred to in the template body. A local definition `name` is bound to a text value `expr` and accessible within the scope of `bodyexpr`:

```
let name = expr  bodyexpr
```

Several let binding can be nested:

```
let name1 = expr1
let name2 = expr2
let name3 = expr3
bodyexpr
```

This is equivalent to:

```
let name1 = expr1
(let name2 = expr2
(let name3 = expr3
bodyexpr ) )
```

since the let-operator is left associative.

Example with several uses of let:

```
template functionDaeOutput2(list<SimEqSystem> nonStateDiscEquations,
                    list<SimEqSystem> removedEquations) ::=
  let &varDecls = buffer ""
  let nonSateDiscPart = (nonStateDiscEquations |> eq =>
           '<%equation_(eq, contextSimulationDescrete, varDecls)%>'
;separator="\n")
  let removedPart = (removedEquations |> eq =>
           '<%equation_(eq, contextSimulationDescrete, varDecls)%>'
;separator="\n")
<<
/* for discrete time variables */
int functionDAE_output2()
{
  state mem_state;
  <%varDecls%>

  mem_state = get_memory_state();
  <%nonSateDiscPart%>
  <%removedPart%>
  restore_memory_state(mem_state);

  return 0;
}
>>
end functionDaeOutput2;
```

12.9.2 let Binding of Buffer Variables and their Use

A buffer variable can be introduced in the following way through a let-binding:

```
let &name = buffer text-expression  body-expression
```

The reference, name, is immutable, but the contents is mutable since it is a buffer. The value of the buffer is initialized to the `text-expression`. As usual, the let-binding of name can be accessed within the scope of the body-expression.

The following language rules apply regarding text buffers

- A text buffer can only be appended to by the += operator in a let expression (Section 12.9.3).
- A text buffer can be passed as an argument to a template function by marking it with & at the call. This is a reference parameter that can be modified by the called function, i.e., an in-out parameter (Section 12.6.6)
- A text buffer can only be referenced inside text template expressions in the template function it is declared in. (text buffers can only be appended to in template functions they are passed to)

12.9.3 Appending a String to a buffer Variable

It is possible to perform a side effect of appending a string `expr` at the end of a buffer variable `var1`:

```
let &var1 =+ expr
```

This is equivalent to the following Modelica code:

```
var1 := var1 + expr;
```

Example 1:

```
let &preExp += 'create_index_spec(&<%tmp%>, <%nridx_str%>, <%idx_str%>);<%\n%>'
```

Example 2:

```
template tempDecl(String ty, Text &varDecls) ::=
  let newVar = 'tmp<%System.tmpTick()%>'
  let &varDecls += '<%ty%> <%newVar%>; <%\n%>'   // varDecls is updated,
appended
  newVar  // return newVar
end tempDecl;
```

12.9.4 Reference (buffer) Formal Parameters in Template Functions

In a template function header, & is used before the reference parameter name, and the type in such cases is always Text.

Example:

```
template funcname(Argtype1 arg1, Text &arg2Reference) ::= ...
```

12.10 Formatting, Separator, and Indentation Options

A number of options can be specified with the option operator to control formatting and indentation of a template expression templ-exp.

`templ-exp ;opt`$_1$`=val`$_1$ `;opt`$_2$ `;opt`$_3$`=val`$_3$ `... ;opt`$_n$`=val`$_n$

It is usually used inside a hole, and can optionally be enclosed in parentheses as any other expression, e.g.:.

`(templ-exp ;opt`$_1$`=val`$_1$ `;opt`$_2$`; ... ;opt`$_n$`=val`$_n$`)`

This is a semi-colon 2n+1-ary left associative operator. All options and option-values are collected, and simultaneously applied to the leftmost operand, the `templ-exp`.

12.10.1 Indentation controlling options

- `anchor` – relative to start of the expression
- `absIndent` – absolutely from the line beginning
- `relIndent` – relative to the actual indent
- `indent` – immediate indent and then relative to the actual indent

12.10.2 Multi-Value Formatting Options

It is *important* to note that these options are only valid for values that contain multiple values, e.g. lists/arrays.

- `separator` – separator text inserted between the results in the multiple-value list/array.
- `align` – the number of results to be aligned by `alignSeparator` – separator for aligning `alignOffset` – start align counting offset.
- `wrap` – number of characters to be wraped by `wrapSeparator` – separator for wrapping.
- `empty` – value substituted for empty results.
- `countEmpty` – *To be implemented* - count empty results when using `hasindex` in an iterator expression
- `separateEmpty` – *To be implemented* – used together with `separator` to control separation of empty results. Default is `false`.

12.10.3 Options

Expression options can be specified only in the direct lexical context of **<** … **>** or (…).

Indentation controlling options control indentation that occurs *before outputting the first non-space character after a new line* inside of the option affected output text.

All indentation options are of type integer where usage without '=' defaults to 0.

The `indent` option outputs the specified number of spaces immediately and then behaves like `relIndent`.

Multi-values formatting options can be applied for all expressions that (possibly) results in concatenation of multiple results, i.e., list or array values, map expressions and (pseudo)list construction expressions.

The `separator` option is used for inserting a separator string between elements of multiple-value expressions t. Default separator value is empty string.

The `align` and `wrap` options are of type integer where a positive value means a number of results or characters, respectively, after which a value of the `alignSeparator` or `wrapSeparator` will be output. Default values of `alignSeparator` and `wrapSeparator` is new-line character. Default value of `align` option is `10` and the default of `wrap` option is `100` (these values are used when the option is specified without a value – i.e., not using '=').

The `alignOffset` option can be used to set the start counting offset from whichthe `align` option counts its effect. Default is 0.

The `empty` option is of string type (more precisely, `Option<String>`), and when specified, its value is used whenever the concatenated result is an empty string. This option has impact on the counting and separation semantics (which is not intuitive in several scenarios). By default, this option is "unset" (internally `NONE`) that means that there is no replacement for empty results and the separation and index counting by default DOES NOT count/separate empty results. When this option is used without a value (without '=' after it), it is equivalent to empty="" (empty string) and it means that "there is a value" for every result, so the counting and separation takes the value as non-empty (it is counted/separated).

To address this complex semantics (a mess), the two options `countEmpty` and `separateEmpty` will be implemented to make the semantics much clearer and explicit. Particularly, this option will only dictate the replacement value for empty result and it will have by default the value of an empty string (not `NONE` as now).

The `countEmpty` – *To be implemented* - option is of Boolean type with default value true (in the current implementation it is false). When set to true, it means that an index variable defined by `hasindex` for this multi-value expression is advanced for empty results, otherwise not. It is important to note that the emptiness is checked against the actual result value regardless of the `empty` option value (so the `empty` option is used effectively only for replacement of the empty results while not affecting the emptiness checks).

See also Section 12.8.1 for more information about `countEmpty` and `hasindex`.

Examples:

```
// note the automatic indentation by 2 spaces
template lines2(list<String> lines) ::= <<
  <%lines ;separator=\n%>
>>
end lines2;


// align by 15 values and anchor the output 1 space after {
template intArr(list<Integer> values) ::= <<
int[] myArr = { <%values ;separator=", " ;align=8 ;anchor%> };
>>
```

```
end intArr;


/* example output:
int[] myArr = { 1, 2, 3, 4, 5, 6, 7, 8,
                9, 10, 11, 12, 13, 15, 16,
                17, 18, 19, 20 };
*/
```

12.11 Interface Packages

Template functions in the Text template language language are grouped in *packages*, currently with file extension `.tpl`. Each template package can import one or more interface packages, i.e., that defines sets of AST type definitions and/or function signatures. Each interface packages uses MetaModelica syntax and resides in a separate `.mo` file.

An interface packages has the same properties as an ordinary Modelica package, but with the following differences:

- Imported items are re-exported the naming they are given depending on the import.
- Imported packages are re-exported with a restricted view – only the items explicitly declared are re-expred.
- Functions in the interface package can only be function headers for giving the function signatures.
- Union types which are re-exported only make the explicitly declared constructors and fields visible in the exported view, i.e., a so-called `type view` for that union type.
- How should an interface package be referred to in other packages (and in the .tpl file?) Answer: with an import statement, e.g. `import interface SimCodeInterface`. Multiple interfaces packages can be imported.

Note that the `public` and `protected` keywords *should not* be used in an interface package before the uniontype and function definitions.

Here we will show an example interface package that models the while loop example from the Modelica'2009 text template paper (Fritzson, Privitzer, Sjölund, and Pop 2009), and defines type views for the `Statement`, `Exp`, and `Operator` union types.

```
interface package InterfacePackageName
...

package OriginalPackageName

uniontype Statement  "Algorithmic stmts"
  record ASSIGN  "An assignment stmt"
    Exp lhs; Exp rhs;
  end ASSIGN;

  record WHILE  "A while statement"
    Exp condition;
    list<Statement> statements;
  end WHILE;
end Statement;

uniontype Exp  "Expression nodes"
  record ICONST  "Integer constant value"
    Integer value;
  end ICONST;

  record VARIABLE "Variable reference"
    String name;
  end VARIABLE;

  record BINARY  "Binary ops"
    Exp lhs; Operator op;  Exp rhs;
```

```
  end BINARY;
end Exp;

uniontype Operator
  record PLUS end PLUS;
  record TIMES end TIMES;
  record LESS end LESS;
end Operator;

end OriginalPackageName;

...
end InterfacePackageName;
```

The `OriginalPackageName` is the name of the original MetaModelica package where types corresponding to the type views in the interface package are fully defined. An interface package can use types from several packages. It usually specifies a subset of the original types defined in several packages and from these types suitable parts can be selected. For example, there can be additional union tags in the `Statement` type, but only those specified in the type view in the interface package can be used by templates that use this view. Similarly, more record fields can be originally defined in the `ASSIGN` record but only `lhs` and `rhs` can be read inside the template package with the view imported.

Interface package files with AST type views can be shared across different target languages as a kind of type interface to the compiler generated output ASTs (e.g., simulation code ASTs). It is also an essential feature to support scenarios where users are not allowed to see all original types (e.g., a commercial Modelica compiler) but still can see and use the intended subset to extend the code generator.

In addition to type views in interface packages, templates automatically understand all MetaModelica built-in types: `String`, `Boolean`, `Integer`, `Real`, `list`, `Option`, `tuple`, and `Array` types.

You can import multiple interface packages.

Example:

```
interface package SimCodeInterface

...

package builtin

  function listLength<TypeVar> "Return the length of the list"
    input list<TypeVar> lst;
    output Integer result;
  end listLength;

end builtin;


package SimCode

  function crefSubIsScalar
    input DAE.ComponentRef cref;
    output Boolean isScalar;
  end crefSubIsScalar;

  ...

  uniontype Context
    record SIMULATION
      Boolean genDiscrete;
    end SIMULATION;
    record OTHER
    end OTHER;
  end Context;
```

```
  constant Context contextSimulationNonDescrete;

  ...
  type Variables = list<Variable>;
  type Statements = list<Statement>;
  type VariableDeclarations = Variables;

  uniontype Variable
    record VARIABLE
      DAE.ComponentRef name;
      Type ty;
      Option<DAE.Exp> value;
      list<DAE.Exp> instDims;
    end VARIABLE;
  end Variable;

  uniontype Statement
    record ALGORITHM
       list<DAE.Statement> statementLst;
    end ALGORITHM;
  end Statement;
...


end SimCode;


package DAELow

  uniontype ZeroCrossing
    record ZERO_CROSSING
      DAE.Exp relation_;
    end ZERO_CROSSING;
  end ZeroCrossing;

...

end DAELow;

...

end SimCodeInterface;
```

12.11.1 Interface Package Import into a Template File

To import the above example interface package into a template file, use the following syntax:

```
import interface SimCodeInterface;
```

By default, all imported symbols (imported types, functions, constants) are also accessible without the need of a package qualification, although it is recommended to visually distinguish from a template call to use proper package names in front of imported functions calls, for example (an extract from SimcodeTV)

```
interface package SimCodeTV
…
package System
…
  function tmpTick
    output Integer tickNo;
  end tmpTick;
…
end System;
…
end SimCodeTV;
```

It is recommended to call this function in template :

```
System.tmpTick()
```

Instead just (working)

```
tmpTick()
```

(*Undocumented feature* - One can force usage of qualified lookup of names of an imported package by the usage of the **protected** keyword in front of the imported package in the interface file, but this usage of the protected keyword is might be redesigned in the future; with ?forcequalified?)

12.12 Template File Import into Another Template File

Importing of a template package into another template package is possible with the **import** statement.

There is *unqualified* and *qualified* import with the semantics similar to (or the same) as in the standard Modelica language. Syntax:

```
import TemplatePackage.*; //unqualified import of all symbols in TemplatePackage
import TemplatePackage;   //fully qualified import of items in TemplatePackage
```

The unqualified import does not require package name paths to be used when referencing an imported symbol. It is still possible to use a (fully-)qualified name when needed, for example when a local symbol hides the imported one (local symbols take precedence over imported ones).

The (fully-)qualified import of a template package forces the usage of fully qualified but shorter paths to referenced imported symbols (now, effectively, only the package name).

12.13 Template Error Handling

Generally, templates should not report any errors as templates are intended only to give a *textual view* of an abstract and correct (already checked) simulation code (or another text view of an abstract representation).

In practice, templates during development are often incomplete with respect to the input structures and an (template-)error mechanism is useful to cover these situations.

A template file can import (via an interface package) two functions from `Tpl` package (Susan's runtime) `Tpl.addSourceTemplateError()` and `Tpl.addTemplateError()` to be able to report an error from within templates. Their signatures are:

```
function addSourceTemplateError
    "Wraps call to Error.addSourceMessage() funtion with Error.TEMPLATE_ERROR
     and one MessageToken."
  input String inErrMsg;
  input Absyn.Info inInfo;
end addSourceTemplateError;

//for completeness; although the addSourceTemplateError() above is preferable
function addTemplateError
  "Wraps call to Error.addMessage() function with Error.TEMPLATE_ERROR and one
   MessageToken."
  input String inErrMsg;
end addTemplateError;
```

The errors reported with `Tpl.addSourceTemplateError()` and `Tpl.addTemplateError()` are checked in the top-level MetaModelica functions `Tpl.tplString()` and `Tpl.noRet()`. These functions are intended to be the only template invocation entry points from MetaModelica code and they will fail when an error was reported inside the processed templates.

In conjunction to these functions, to be able to also report the source location of a template error, a built-in function `sourceInfo()`is introduced that (magically) returns the contextual `Absyn.Info` structure based on the position in the template file where it is "invoked". The `sourceInfo()` can only

be used as a direct parameter into a template/imported function. No let binding of its value is currently possible (it will be possible when the full semantics of the let will be implemented).

Then, user can create his/her own error reporting templates. For example for a C++ generator

```
template error(Absyn.Info srcInfo, String errMessage)
 "Example source template error reporting template to be used together with the
 sourceInfo() magic function.
 Usage: error(sourceInfo(), <<message>>)"
::=
let() = Tpl.addSourceTemplateError(errMessage, srcInfo)
<<

#error "<% Error.infoStr(srcInfo) %> <% errMessage %>"<%\n%>
>>
end error;
```

(Note: the `Error.infoStr()` function must be imported via an interface package)

The following example uses the above template:

```
template literalExpConstBoxedVal(Exp lit) ::=
  match lit
  case ICONST(__) then 'MMC_IMMEDIATE(MMC_TAGFIXNUM(<%integer%>))'
  case BCONST(__) then 'MMC_IMMEDIATE(MMC_TAGFIXNUM(<%bool%>))'
  //… original cut for brevity
  case lit as SHARED_LITERAL(__) then '_OMC_LIT<%lit.index%>'
  else error(
        sourceInfo(),
       'literalExpConstBoxedVal failed: <%printExpStr(lit)%>')
end literalExpConstBoxedVal;
```

Appendix A

MetaModelica Grammar

This appendix contains the grammar of the MetaModelica 2.0 language.

A.1 Class and Main Grammar Elements

```
 stored_definition :
  ( within_clause SEMICOLON )?
  class_definition_list?
  EOF
  ;

within_clause :
    WITHIN ( name_path )?
  ;

class_definition_list :
  ( (FINAL)? class_definition SEMICOLON ) class_definition_list?
  ;

class_definition :
  ( (ENCAPSULATED)? (PARTIAL)? class_type class_specifier )
  ;

class_type :
  ( CLASS
  | OPTIMIZATION
  | MODEL
  | RECORD
  | BLOCK
  | ( EXPANDABLE )? CONNECTOR
  | TYPE
  | T_PACKAGE
  | FUNCTION
  | UNIONTYPE
  | OPERATOR (FUNCTION | RECORD)?
  )
  ;

identifier :
  (IDENT|DER|CODE|EQUALITY|INITIAL)
  ;

class_specifier :
    ( identifier class_specifier2
    | EXTENDS identifier (class_modification)? string_comment composition T_END identifier
    )
    ;

class_specifier2 :
  (
    ( LESS ident_list GREATER )? string_comment composition T_END identifier
  | EQUALS base_prefix type_specifier ( class_modification )? comment
  | EQUALS enumeration
  | EQUALS pder
  | EQUALS overloading
  )
  ;

pder :
```

```
  DER LPAR name_path COMMA ident_list RPAR comment
  ;

ident_list :
  IDENT (COMMA ident_list)?
  ;


overloading :
  OVERLOAD LPAR name_list RPAR comment
  ;

base_prefix :
  type_prefix
  ;

name_list :
  name_path (COMMA name_list)?
  ;

enumeration :
  ENUMERATION LPAR (enum_list | COLON ) RPAR comment
  ;

enum_list :
  enumeration_literal ( COMMA enum_list )?
  ;

enumeration_literal :
  IDENT comment
  ;

composition :
  element_list composition2
  ;

composition2 :
  ( external_clause?
  | ( public_element_list
    | protected_element_list
    | initial_equation_clause
    | initial_algorithm_clause
    | equation_clause
    | constraint_clause
    | algorithm_clause
    ) composition2
  )
  ;

external_clause :
  EXTERNAL
  ( language_specification )?
  ( ( component_reference EQUALS )?
    IDENT LPAR ( expression_list )? RPAR )?
  ( annotation )? SEMICOLON
  ( external_annotation )?
  ;

external_annotation :
  annotation SEMICOLON
  ;

public_element_list :
  PUBLIC element_list
  ;

protected_element_list :
  PROTECTED element_list
  ;

language_specification :
  STRING
  ;

element_list :
  ( ( ( element | annotation ) SEMICOLON ) element_list )?
```

```
  ;

element :
    import_clause
  | extends_clause
  | (REDECLARE)? (FINAL)? (INNER)? (T_OUTER)?
    ( ( class_definition | component_clause )
    | (REPLACEABLE ( class_definition | component_clause ) constraining_clause_comment? )
    )
  ;

import_clause :
  IMPORT ( explicit_import_name | implicit_import_name ) comment
  ;

explicit_import_name :
  (IDENT|CODE) EQUALS name_path
  ;

implicit_import_name :
  name_path_star
  ;
```

A.2 Extends

```
/*
 * 2.2.3 Extends
 */

// Note that this is a minor modification of the standard by
// allowing the comment.
extends_clause :
  EXTENDS name_path (class_modification)? (annotation)?
  ;

constraining_clause_comment :
  constraining_clause comment
  ;

constraining_clause :
    EXTENDS name_path  (class_modification)?
  | CONSTRAINEDBY name_path (class_modification)?
  ;

/*
 * 2.2.4 Component clause
 */

component_clause :
  type_prefix type_specifier component_list
  ;

type_prefix :
  (FLOW|STREAM)? (DISCRETE|PARAMETER|CONSTANT)? (T_INPUT|T_OUTPUT)?
  ;

type_specifier :
  name_path (LESS type_specifier_list GREATER)? (array_subscripts)?
  ;

type_specifier_list :
  type_specifier (COMMA type_specifier_list)?
  ;

component_list :
  component_declaration (COMMA component_list)?
  ;

component_declaration :
  declaration (conditional_attribute)? comment
  ;

conditional_attribute :
  IF expression
  ;
```

```
declaration :
  ( IDENT | OPERATOR ) (array_subscripts)? (modification)?
  ;
```

A.3 Modification

```
/*
 * 2.2.5 Modification
 */

modification :
  ( class_modification ( EQUALS expression )?
  | EQUALS expression
  | ASSIGN expression
  )
  ;

class_modification :
  LPAR ( argument_list )? RPAR
  ;

argument_list :
  argument ( COMMA argument_list )?
  ;

argument :
  ( element_modification_or_replaceable
  | element_redeclaration
  )
  ;

element_modification_or_replaceable :
  (EACH)? (FINAL)? (element_modification | element_replaceable)
  ;

element_modification :
  component_reference ( modification )? string_comment
  ;

element_redeclaration :
  REDECLARE (EACH)? (FINAL)?
  ( (class_definition | component_clause1) | element_replaceable )
  ;

element_replaceable :
  REPLACEABLE ( class_definition[final] | component_clause1 ) constraining_clause_comment?
  ;

component_clause1 :
  base_prefix type_specifier component_declaration1
  ;

component_declaration1 :
  declaration comment
  ;
```

A.4 Equations

```
/*
 * 2.2.6 Equations
 */

initial_equation_clause :
  { LA(2) == EQUATION }?
  INITIAL EQUATION equation_annotation_list
  ;

equation_clause :
  EQUATION equation_annotation_list
  ;

constraint_clause :
  CONSTRAINT constraint_annotation_list
```

```
  ;

equation_annotation_list :
  { LA(1) == T_END || LA(1) == CONSTRAINT || LA(1) == EQUATION || LA(1) == T_ALGORITHM ||
    LA(1) == INITIAL || LA(1) == PROTECTED || LA(1) == PUBLIC }?
  | ( equation SEMICOLON | annotation SEMICOLON ) equation_annotation_list
  ;

constraint_annotation_list :
  { LA(1) == T_END || LA(1) == CONSTRAINT || LA(1) == EQUATION || LA(1) == T_ALGORITHM ||
    LA(1) == INITIAL || LA(1) == PROTECTED || LA(1) == PUBLIC }?
  | ( equation SEMICOLON | annotation SEMICOLON ) constraint_annotation_list
  ;

algorithm_clause :
  T_ALGORITHM algorithm_annotation_list
  ;

initial_algorithm_clause :
  { LA(2)==T_ALGORITHM }?
  INITIAL T_ALGORITHM algorithm_annotation_list
  ;

algorithm_annotation_list :
  { LA(1) == T_END || LA(1) == EQUATION || LA(1) == T_ALGORITHM ||
    LA(1) == INITIAL || LA(1) == PROTECTED || LA(1) == PUBLIC }?
  | ( algorithm SEMICOLON | annotation SEMICOLON ) algorithm_annotation_list
  ;

equation :
  ( equality_or_noretcall_equation
  | conditional_equation_e
  | for_clause_e
  | connect_clause
  | when_clause_e
  | FAILURE LPAR equation RPAR
  | EQUALITY LPAR expression EQUALS expression RPAR
  )
  comment
  ;

constraint :
  ( equality_or_noretcall_equation
  | conditional_equation_e
  | for_clause_e
  | connect_clause
  | when_clause_e
  | FAILURE LPAR equation RPAR
  | EQUALITY LPAR expression EQUALS expression RPAR
  )
  comment
  ;


algorithm :
  ( assign_clause_a
  | conditional_equation_a
  | for_clause_a
  | while_clause
  | when_clause_a
  | BREAK
  | RETURN
  | FAILURE LPAR algorithm RPAR
  | EQUALITY LPAR expression ASSIGN expression RPAR
  )
  comment
  ;

assign_clause_a :
  simple_expression ( ( ASSIGN | eq = EQUALS ) expression )?
  ;

equality_or_noretcall_equation :
  simple_expression
    (  ( EQUALS | ASSIGN ) expression
    |  { LA(1) != EQUALS && LA(1) != ASSIGN }?
```

```
    )
  ;

conditional_equation_e :
  IF expression THEN equation_list equation_elseif_list? ( ELSE equation_list )? T_END IF
  ;

conditional_equation_a :
  IF expression THEN algorithm_list algorithm_elseif_list? ( ELSE algorithm_list )? T_END IF
  ;

for_clause_e :
  FOR for_indices LOOP equation_list T_END FOR
  ;

for_clause_a :
  FOR for_indices LOOP algorithm_list T_END FOR
  ;

while_clause :
  WHILE expression LOOP algorithm_list T_END WHILE
  ;

when_clause_e :
  WHEN expression THEN equation_list else_when_e_list? T_END WHEN
  ;

else_when_e_list :
  else_when_e else_when_e_list?
  ;

else_when_e :
  ELSEWHEN expression THEN equation_list
  ;

when_clause_a :
  WHEN expression THEN algorithm_list else_when_a_list? T_END WHEN
  ;

else_when_a_list :
  else_when_a else_when_a_list?
  ;

else_when_a :
  ELSEWHEN expression THEN algorithm_list
  ;

equation_elseif_list :
  equation_elseif equation_elseif_list?
  ;

equation_elseif :
  ELSEIF expression THEN equation_list
  ;

algorithm_elseif_list :
  algorithm_elseif algorithm_elseif_list?
  ;

algorithm_elseif :
  ELSEIF expression THEN algorithm_list
  ;

equation_list_then :
    { LA(1) == THEN }?
  | (equation SEMICOLON equation_list_then)
  ;


equation_list :
    { LA(1) != T_END || (LA(1) == T_END && LA(2) != IDENT) }?
  | ( equation SEMICOLON equation_list )
  ;

algorithm_list :
    { LA(1) != T_END || (LA(1) == T_END && LA(2) != IDENT) }?
```

```
  | algorithm SEMICOLON algorithm_list
  ;

connect_clause :
  CONNECT LPAR component_reference COMMA component_reference RPAR
  ;
```

A.5 Expressions

```
/*
 * 2.2.7 Expressions
 */
expression :
  ( if_expression
  | simple_expression
  | part_eval_function_expression
  | match_expression
  )
  ;

part_eval_function_expression :
  FUNCTION component_reference function_call
  ;

if_expression :
  IF expression THEN expression elseif_expression_list? ELSE expression
  ;

elseif_expression_list :
  elseif_expression elseif_expression_list?
  ;

elseif_expression :
  ELSEIF expression THEN expression
  ;

for_indices :
  for_index (COMMA for_indices)?
  ;

for_index :
  (IDENT ((GUARD expression)? T_IN expression)?)
  ;

simple_expression :
    simple_expr (COLONCOLON simple_expression)?
  | IDENT AS simple_expression
  ;

simple_expr :
  logical_expression ( COLON logical_expression ( COLON logical_expression )? )?
  ;

logical_expression :
  logical_term ( T_OR logical_term )*
  ;

logical_term :
  logical_factor ( T_AND logical_factor )*
  ;

logical_factor :
  ( T_NOT )? relation
  ;

relation :
  arithmetic_expression
  ( ( LESS | LESSEQ | GREATER | GREATEREQ | EQEQ | LESSGT ) arithmetic_expression )?
  ;

arithmetic_expression :
  unary_arithmetic_expression ( ( PLUS | MINUS | PLUS_EW | MINUS_EW ) term )*
  ;

unary_arithmetic_expression :
```

```
  ( PLUS term
  | MINUS term
  | PLUS_EW term
  | MINUS_EW term
  | term
  )
  ;

term :
  factor ( ( STAR | SLASH | STAR_EW | SLASH_EW ) factor )*
  ;

factor :
  primary ( ( POWER | POWER_EW ) primary )?
  ;

primary :
  ( UNSIGNED_INTEGER
  | UNSIGNED_REAL
  | STRING
  | T_FALSE
  | T_TRUE
  | component_reference__function_call
  | DER function_call
  | LPAR output_expression_list
  | LBRACK matrix_expression_list RBRACK
  | LBRACE for_or_expression_list RBRACE
  | T_END
  )
  ;

matrix_expression_list :
  expression_list (SEMICOLON matrix_expression_list)?
  ;

component_reference__function_call :
  component_reference ( function_call )?
  | INITIAL LPAR RPAR
  ;

name_path :
  (DOT)? name_path2
  ;

name_path2 :
    { LA(2) != DOT }? (IDENT | CODE)
  | (IDENT | CODE) DOT name_path
  ;

name_path_star :
  (DOT)? name_path_star2
  ;

name_path_star2 :
    { LA(2) != DOT }? (IDENT | CODE) ( STAR_EW )?
  | (IDENT | CODE) DOT name_path_star2
  ;

component_reference :
  (DOT)? component_reference2
  | ALLWILD
  | WILD
  ;

component_reference2 :
    (IDENT | OPERATOR) ( array_subscripts )? ( DOT component_reference2 )?
  ;

function_call :
  LPAR function_arguments RPAR
  ;

function_arguments :
  for_or_expression_list (named_arguments) ?
  ;
```

```
for_or_expression_list :
  ( { LA(1) == IDENT || LA(1) == OPERATOR && LA(2) == EQUALS ||
      LA(1) == RPAR  || LA(1) == RBRACE }?
  | ( expression
      ( (COMMA for_or_expression_list2)
      | (FOR for_indices)
      )?
    )
  )
  ;

for_or_expression_list2 :
    {LA(2) == EQUALS}?
  | expression (COMMA for_or_expression_list2)?
  ;

named_arguments :
  named_argument (COMMA named_arguments)?
  ;

named_argument :
  ( IDENT | OPERATOR) EQUALS expression
  ;

output_expression_list :
  ( RPAR
  | COMMA output_expression_list
  | expression
    ( COMMA output_expression_list
    | RPAR
    )
  )
  ;

expression_list :
  expression (COMMA expression_list)?
  ;

array_subscripts :
  LBRACK subscript_list RBRACK
  ;

subscript_list :
  subscript ( COMMA subscript_list )?
  ;

subscript :
    expression
  | COLON
  ;

comment :
  (string_comment (annotation)?)
  ;

string_comment :
  ( STRING (PLUS s2 = STRING)* )?
  ;

annotation :
  T_ANNOTATION class_modification
  ;
```

A.6 MetaModelica Extensions

```
/* MetaModelica */
match_expression :
  ( (MATCHCONTINUE expression string_comment
     local_clause
     cases
     T_END MATCHCONTINUE)
  | (MATCH expression string_comment
     local_clause
     cases
     T_END MATCH)
```

```
  )
  ;

local_clause :
  (LOCAL el=element_list)?
  ;

cases :
  onecase cases2
  ;

cases2 :
  ( (ELSE (string_comment (EQUATION equation_list_then)? THEN)? expression SEMICOLON)?
  | onecase cases2
  )
  ;

onecase :
  CASE pattern string_comment (EQUATION equation_list_then)? THEN expression SEMICOLON
  ;

pattern :
  e=expression
  ;
```

Appendix B

Predefined MetaModelica Operators and Functions

This appendix contains a number of basic primitives, for which the semantics is assumed to be known or having a mathematical definition. First we show the precedence and associativity of the builtin operators. Then we present short definitions of the builtin functions. Finally we present the definitions packaged in a standard MetaModelica package, called `MetaModelica`, shown below. First the type signatures of all predefined primitives are presented. Then the semantics of some primitives is explained or defined in MetaModelica.

B.1 Precedence of Predefined Operators

The following table presents all the operators in order of precedence from highest to lowest. All operators are binary except the postfix operators and those shown as unary together with *expr*, the conditional operator, the array construction operator {} and . Operators with the same precedence occur at the same line of the table:

Table B-1. Operators and their precedence

Operator Group	*Operator Syntax*	*Examples*
postfix index operator	`[]`	`arr[index]`
name dot notation	`.`	`PackageA.func`
postfix function call	(*function-arguments*)	`sin(4.36)`
array or list construction	{*expressions*} `List`(expressions)	`{2,3}`
rpower of	`^`	`x ^ 2`
Multiplicative	`* /`	`2*3 2/3.2 a*b a/b`
Additive	`+ - +`*expr* `-`*expr*	`a+b, a-b, +a, -a`
Relational	`< <= > >= == <>`	`a<b, a<=b, a>b, ...`
unary negation	`not` *expr*	`not b1`
logical and	`and`	`b1 and b2`
logical or	`or`	`b1 or b2`
conditional expression	`if` *expr* `then` *expr* `else` *expr*	`if b then 3 else x`
list element concatenation	`"a"::{"b","c"}` => `{"a","b","c"}`	`"a"::{"b","c"}` => `{"a","b","c"}`
named argument	*ident* `=` *expr*	`x = 2.26`

Equality `=` and assignment `:=` are not expression operators since they are allowed only in equations and in assignment statements respectively. All binary expression operators are left associative. There is also a generic structural equality operator, `valueEq(expr1,expr2)`, giving `false` or `true`, which can be applied to values of primitive data types as well as to values of structured types such as arrays, lists, trees or any other user defined type.

B.2 Short Descriptions of Builtin Functions and Operators

In this section we provide approximate or exact short descriptions of the builtin MetaModelica primitive functions and operators. Most functions are defined in terms of known mathematical operators.

- `clock()` – return a clock tick value.

- `print(TypeVar)` – print a value.
- `tick()` – generate a unique integer compared previous calls to tick from the start of this execution.

The following operations only apply to primitive MetaModelica values, which can be either an integer `i`, a real `r`, a string `str`, a list `lst`, an array `vec`, or an unbound location.

- `intAdd(i1,i2)` = `i1` + `i2` if the result can be represented by the implementation, otherwise the operation fails.
- `intSub(i1,i2)` = `i1` – `i2` if the result can be represented by the implementation, otherwise the operation fails.
- `intMul(i1,i2)` = `i1` × `i2` if the result can be represented by the implementation, otherwise the operation fails.
- `intDiv(i1,i2)` returns the Real quotient of `i1` and `i2` if `i2` ≠ 0 and the result can be represented by the implementation, otherwise the operation fails.
- `intMod(i1,i2)` returns the integer modulus of `i1` and `i2` if `i2` ≠ 0 and the result can be represented by the implementation, otherwise the operation fails.
- `intAbs(i)` returns the absolute value of `i` if the result can be represented by the implementation, otherwise the operation fails.
- `intNeg(i)` returns –`i` if the result can be represented by the implementation, otherwise the
- operation fails.
- `intMax(i1,i2)` = `i1` if `i1` ≥ `i2`, otherwise `i2`.
- `intMin(i1,i2)` = `i1` if `i1` ≤ `i2`, otherwise `i2`.
- `intLt(i1,i2)` = `true` if `i1` < `i2`, otherwise `false`.
- `intLe(i1,i2)` = `true` if `i1` ≤ `i2`, otherwise `false`.
- `intEq(i1,i2)` = `true` if `i1` = `i2`, otherwise `false`.
- `intNe(i1,i2)` = `true` if `i1` ≠ `i2`, otherwise `false`.
- `intGe(i1,i2)` = `true` if `i1` ≥ `i2`, otherwise `false`.
- `intGt(i1,i2)` = `true` if `i1` > `i2`, otherwise `false`.
- `intReal(i)` = `r` where `r` is the corresponding real value equal to `i`.
- `intString(i)` returns a textual representation of `i`, as a string.
- `realAdd(r1,r2)` = `r1` + `r2`.
- `realSub(r1,r2)` = `r1` – `r2`.
- `realMul(r1,r2)` = `r1` × `r2`.
- `realDiv(r1,r2)` = `r1` / `r2`.
- `realMod(r1,r2)` = returns the integer modulus of `r1` / `r2`. This is the value `r1` – `i` × `r2`, for some integer `i` such that the result has the same sign as `r1` and magnitude less than the magnitude of `r2`. If `r2` = `0`, the operation fails.
- `realAbs(r)` returns the absolute value of `r`.
- `realNeg(r)` = –`r`.
- `realCos(r)` returns the cosine of `r` (measured in radians).
- `realSin(r)` returns the sine of `r` (measured in radians).
- `realAtan(r)` returns the arc tangent of `r`.
- `realExp(r)` returns e^r .
- `realLn(r)` returns ln(`r`).
- `realFloor(r)` returns the largest integer (as a real value) not greater than `r`.
- `realInt(r)` discards the fractional part of `r` and returns the integral part as an integer; fails if this value cannot be represented by the implementation.

- `realPow(r1,r2)` = $r1^{r2}$; fails if this cannot be computed.
- `realSqrt(r)` = $\sqrt{r}$; fails if r < 0.
- `realMax(r1,r2)` = `r1` if `r1` ≥ `r2`, otherwise `r2`.
- `realMin(r1,r2)` = `r1` if `r1` ≤ `r2`, otherwise `r2`.
- `realLt(r1,r2)` = `true` if `r1` < `r2`, otherwise `false`.
- `realLe(r1,r2)` = `true` if `r1` ≤ `r2`, otherwise `false`.
- `realEq(r1,r2)` = `true` if `r1` = `r2`, otherwise `false`.
- `realNe(r1,r2)` = `true` if `r1` ≠ `r2`, otherwise `false`.
- `realGe(r1,r2)` = `true` if `r1` ≥ `r2`, otherwise `false`.
- `realGt(r1,r2)` = `true` if `r1` > `r2`, otherwise `false`.
- `stringInt(str)` = `i` if the string `str` has the lexical structure of an integer constant and `i` is the value associated with that constant. Otherwise the operation fails.

B.3 Interface to the Standard MetaModelica Package

The following subsections present type signatures for all builtin MetaModelica primitives. First comes the package header for the `MetaModelica` package:

```
package MetaModelica:
```

B.3.1 Predefined Types and Type Constructors

The following predefined types are available:

```
type Integer "Builtin Integer type";
type Real    "Builtin Real type";
type String  "Builtin String type";

uniontype Boolean  "Builtin Boolean type"
  record false end false;
  record true  end true;
end Boolean;

uniontype List<TypeVar>  "Builtin List uniontype"
  record nil "the nil constructor" end nil;
  record cons "the cons constructor, also ::"
    TypeVar       head "The head of the list";
    List<TypeVar> tail "The rest of the list";
  end cons;
end List;

uniontype Option<TypeVar> "Builtin Option uniontype"
  record NONE "the NONE constructor" end NONE;
  record SOME "the SOME constructor"
    TypeVar some;
  end SOME;
end Option;
```

B.3.2 Boolean Operations

The following builtin boolean operations are available:

```
function boolAnd "Boolean and"
  input Boolean b1;
  input Boolean b2;
  output Boolean result;
algorithm
  result := match (b1, b2)
     case (true, true) then true;
     case (_, _) then false;
```

```
  end match;
end boolAnd;

function boolOr "Boolean or"
  input Boolean b1;
  input Boolean b2;
  output Boolean result;
algorithm
  result := match (b1, b2)
    case (true, _) then true;
    case (_, true) then true;
    case (_, _) then false;
  end match;
end boolOr;

function boolNot "Boolean not"
   input Boolean b;
   output Boolean result;
algorithm
  result := match (b)
    case (true)  then false;
    case (false) then true;
  end match;
end boolNot;
```

B.3.3 Integer Operations

Some of the builtin integer operations are also available as operators according to the following table:

```
intAdd        +
intSub        -
intNeg        -      // unary -
intMul        *
intDiv        /
intEq         ==
intNe         <>
intGe         >=
intGt         >
intLe         <=
intLt         >
```

The following builtin integer operations are available:

```
function intAdd "Integer addition"
  input Integer i1;
  input Integer i2;
  output Integer result;
algorithm
  result := i1 + i2;
end intAdd;

function intSub "Integer subtraction"
  input Integer i1;
  input Integer i2;
  output Integer result;
algorithm
  result := i1 - i2;
end intSub;

function intMul "Integer multiplication"
  input Integer i1;
  input Integer i2;
  output Integer result;
algorithm
  result := i1 * i2;
end intMul;

function intDiv "Integer division"
  input Integer i1;
```

```
  input Integer i2;
  output Integer result;
algorithm
  result := div(i1, i2);
end intDiv;

function intMod "Integer modulus"
  input Integer i1;
  input Integer i2;
  output Integer result;
algorithm
  result := mod(i1, i2);
end intMod;

function intAbs "Absolute value of the integer"
  input Integer i;
  output Integer result;
algorithm
  result := if (i < 0) then -i else i;
end intAbs;

function intNeg "Integer negation"
  input Integer i;
  output Integer result;
algorithm
  result := -i;
end intNeg;

function intMax "Returns the maximmum of the two integers"
  input Integer i1;
  input Integer i2;
  output Integer result;
algorithm
  result := if i1 >= i2 then i1 else i2;
end intMax;

function intMin "Returns the minimum of the two integers"
  input Integer i1;
  input Integer i2;
  output Integer result;
algorithm
  result := if i1 <= i2 then i1 else i2;
end intMin;

function intLt "Less than integer comparison"
  input Integer i1;
  input Integer i2;
  output Boolean result;
algorithm
  result := (i1 < i2);
end intLt;

function intLt "Less or equal integer comparison"
  input Integer i1;
  input Integer i2;
  output Boolean result;
algorithm
  result := (i1 <= i2);
end intLt;

function intEq "Integer equality comparison"
  input Integer i1;
  input Integer i2;
  output Boolean result;
algorithm
  result := (i1 == i2);
end intEq;

function intNe "Different than integer comparison"
  input Integer i1;
  input Integer i2;
```

```
  output Boolean result;
algorithm
  result := (i1 <> i2);
end intNe;

function intGe "Greater or equal integer comparison"
  input Integer i1;
  input Integer i2;
  output Boolean result;
algorithm
  result := (i1 >= i2);
end intGe;

function intGt "Greater than integer comparison"
  input Integer i1;
  input Integer i2;
  output Boolean result;
algorithm
  result := (i1 > i2);
end intGt;

function intReal "Integer to Real conversion"
  input Integer i;
  output Real result;
algorithm
  result := (Real)i;
end intReal;

function intString "Integer to String conversion"
  input Integer i;
  output String result;
algorithm
  // convert 'i' to String and assign it to the 'result'
end intString;
```

B.3.4 Real Number Operations

Some of the builtin `Real` number operations are also available as overloaded operators according to the following table:

```
realAdd       +
realSub       -
realNeg       -   // unary
realMul       *
realDiv       /
realPow       ^
realEq        ==
realNe        <>
realGe        >=
realGt        >
realLe        <=
realLt        >
```

The following builtin real number operations are available:

```
function realAdd "Real addition"
  input Real r1;
  input Real r2;
  output Real result;
algorithm
  result := r1 + r2;
end realAdd;

function realSub "Real subtraction"
  input Real r1;
  input Real r2;
  output Real result;
algorithm
  result := r1 - r2;
```

```
end realSub;

function realMul "Real multiplication"
  input Real r1;
  input Real r2;
  output Real result;
algorithm
  result := r1 * r2;
end realMul;

function realDiv "Real division"
  input Real r1;
  input Real r2;
  output Real result;
algorithm
  result := r1 / r2;
end realDiv;

function realMod "Real modulo"
  input Real r1;
  input Real r2;
  output Real result;
algorithm
  result := mod(r1, r2);
end realMod;

function realAbs "Absolute value of the Real"
  input Real r;
  output Real result;
algorithm
  result := if (r < 0.0) then -r else r;
end realAbs;

function realNeg "Real negation"
  input Real r;
  output Real result;
algorithm
  result := -r;
end realNeg;

function realCos "Cosine"
  input Real r;
  output Real result;
algorithm
  result := cos(r);
end realCos;

function realSin "Sine"
  input Real r;
  output Real result;
algorithm
  result := sin(r);
end realSin;

function realAtan "Arcustangent"
  input Real r;
  output Real result;
algorithm
  result := atan(r);
end realAtan;

function realExp "Exponentiation"
  input Real r;
  output Real result;
algorithm
  result := exp(r);
end realExp;

function realLn "Natural logarithm"
  input Real r;
  output Real result;
```

```
algorithm
  result := ln(r);
end realLn;

function realFloor "Floor of the real"
  input Real r;
  output Real result;
algorithm
  result := floor(r);
end realFloor;

function realPow "Power r^p"
  input Real r;
  input Real p;
  output Real result;
algorithm
  result := r ^ p;  // r to the power of p
end realPow;

function realSqrt "Square root of the real"
  input Real r;
  output Real result;
algorithm
  result := sqrt(r);
end realSqrt;

function realMax "Returns the maximmum of the two Reals"
  input Real i1;
  input Real i2;
  output Real result;
algorithm
  result := if i1 >= i2 then i1 else i2;
end realMax;

function realMin "Returns the minimum of the two Reals"
  input Real i1;
  input Real i2;
  output Real result;
algorithm
  result := if i1 <= i2 then i1 else i2;
end realMin;

function realLt "Less than Real comparison"
  input Real i1;
  input Real i2;
  output Boolean result;
algorithm
  result := (i1 < i2);
end realLt;

function realLe "Less or equal Real comparison"
  input Real i1;
  input Real i2;
  output Boolean result;
algorithm
  result := (i1 <= i2);
end realLe;

function realEq "Real equality comparison"
  input Real i1;
  input Real i2;
  output Boolean result;
algorithm
  result := (i1 == i2);
end realEq;

function realNe "Different than Real comparison"
  input Real i1;
  input Real i2;
  output Boolean result;
algorithm
```

```
  result := (i1 <> i2);
end realNe;

function realGe "Greater or equal Real comparison"
  input Real i1;
  input Real i2;
  output Boolean result;
algorithm
  result := (i1 >= i2);
end realGe;

function realGt "Greater than Real comparison"
  input Real i1;
  input Real i2;
  output Boolean result;
algorithm
  result := (i1 > i2);
end realGt;

function realInteger "Real to Integer conversion"
  input Real i;
  output Integer result;
algorithm
  // result := (Integer)i;
end realInteger;

function realString "Real to String conversion"
  input Real i;
  output String result;
algorithm
 // convert 'i' to String and assign it to the 'result';
end realString;
```

B.3.5 String Character Conversion Operations

The following builtin conversion operations between one character strings and integer ascii codes are available:

```
function stringCharInt "Returns string character ascii code as integer"
  input String ch;
  output Integer i;
algorithm
  // return the ascii code of the string character
end stringCharInt;

function intStringChar "Returns string char with the given ascii code"
  input Integer i;
  output String ch;
algorithm
  // return string character with the given ascii code
end intStringChar;
```

B.3.6 String Operations

Two builtin operations for `String` are also available as operators according to the following table:

```
stringAppend      +
stringEqual       ==
```

The following builtin String operations are available:

```
function stringInt "Transforms the numerical string value into an integer value"
  input String s;
  output Integer result;
algorithm
  // return the integer representing the
  // string or fail if no such integer exists
end stringInt;
```

```
function stringListStringChar
  "Explode the string to a list of string characters"
  input String s;
  output List<String> listOfStringChars;
algorithm
  // make a list with all the string characters as 1-char strings
end stringListStringChar;

function stringCharListString "From a list of string characters build a string"
  input List<String> listOfStringChars;
  output String s;
algorithm
  // make a string with all string characters in the list
end stringCharListString;

function stringLength "Return the length of the string"
  input String s;
  output Integer result;
algorithm
  // return the length of the given string
end stringLength;

function stringGetStringChar
  "Return the string character from the given string
   at the given index. The string indexing starts from 1."
  input String s;
  input Integer index;
  output String ch;
algorithm
  // Return the string character from the given string at the given index.
  // The string indexing starts from 1.
end stringGetStringChar;

function stringAppend
  "Return a new string with the first string
   appended to the second string given"
  input String s1;
  input String s2;
  output String result;
algorithm
 // Return a new string with the first string appended to second string given
end stringAppend;

function stringUpdateStringChar
  "Return the string given by replacing the string
   char at the given index with the given string char"
  input String s;
  input String ch;
  input Integer index;
  output String result;
algorithm
  // Return the string given by replacing the string char at
  // the given index with the given string char.
  // The string indexing starts from 1.
end stringUpdateStringChar;

function stringEqual "Compares two strings"
  input String s1;
  input String s2;
  output Boolean result;
algorithm
  // true if s1==s2, false otherwise.
end stringEqual;

function stringCompare "Compares two strings"
  input String s1;
  input String s2;
  output Integer result;
algorithm
```

```
  // result = strcmp(s1, s2); using the C language strcmp
end stringCompare;
```

B.3.7 List Operations

The following builtin list operations are available. The operations are polymorphic since they operate on lists of any element type. The replaceable type variable TypeVar is inferred from the input argument types.

```
function listAppend<TypeVar> "Appends two lists and returns the result"
  input List<TypeVar> l1;
  input List<TypeVar> l2;
  output List<TypeVar> result;
algorithm
  // append l1 to l2 and return the result as a new list
end listAppend;

function listReverse<TypeVar> "Reverse the order of elements in the list"
  input List<TypeVar> lst;
  output List<TypeVar> result;
algorithm
  // reverse the order in lst and return the result as a new list
end listReverse;

function listLength<TypeVar> "Return the length of the list"
  input List<TypeVar> lst;
  output Integer result;
algorithm
  // count the elements in the list and return the result
end listLength;

function listMember<TypeVar> "Verify if an element is part of the list"
  input TypeVar element;
  input List<TypeVar> lst;
  output Boolean result;
algorithm
  // return true if the element belongs to the list, false otherwise
end listMember;

function listGet<TypeVar>
  "Return the element of the list at the given index.
   The index starts from 1."
  input List<TypeVar> lst;
  input Integer index;
  output TypeVar result;
algorithm
  // return the element of the list at the index position.
end listGet;

function listDelete<TypeVar>
  "Return a new list without the element at the
   given index. The index starts from 1."
  input List<TypeVar> lst;
  input Integer index;
  output List<TypeVar> result;
algorithm
  // Return a new list without the element at the given index.
  //   The index starts from 1.
end listDelete;
```

B.3.8 MRArray Operations

The `mrarrayGet` function is equivalent to the array indexing operator which is overloaded to work on MRArrays as well as standard Modelica arrays:

```
mrarrayGet(arr, index)  <=>  arr[index]
```

The standard Modelica `fill` function is similar to `mrarrayCreate`:

```
newarr = fill(v, n);   mnewarr = mrarrayCreate(n, v);
```

The following builtin array operations are available:

```
function mrarrayLength<TypeVar> "Returns the length of the array"
  input  MRArray<TypeVar> arr;
  output Integer result;
algorithm
 // Returns the length of the array
end mrarrayLength;

function mrarrayGet<TypeVar> "Returns the element of the array at the given
                             index.  The index starts at 1. "
  input MRArray<TypeVar> arr;
  input Integer index;
  output TypeVar result;
algorithm
 result := arr[index];
end mrarrayGet;

function mrarrayNArray<TypeVar> "Return the elements of the possibly nested
mrarray as a possibly multidimensional array"
  input MRArray<TypeVar> arr;
  output TypeVar[:] result;
algorithm
 // return the elements of the mrarray recursively as a rectangular array.
 // If mrarray is nested, the sub-arrays must be of equal length/size
end mrarrayNArray;

function mrarrayArray<TypeVar> "Return the elements of the mrarray as an array"
  input MRArray<TypeVar> arr;
  output TypeVar[:] result;
algorithm
 // return the elements of the mrarray as a standard rectangular array
 // If mrarray is nested, the sub-arrays must be of equal length/size
end mrarrayArray;

function arrayNMRArray<TypeVar> "Returns the elements of the possibly nested
                                 array as an MRArray"
  input TypeVar[:] arr;
  output MRArray<TypeVar> result;
algorithm
 // return the elements of the possibly nested array as an MRArray
end arrayNMRArray;

function arrayMRArray<TypeVar> "Returns the elements of the array as an MRArray"
  input TypeVar[:] arr;
  output MRArray<TypeVar> result;
algorithm
 // return the elements of the list as an array
end arrayMRArray;

function mrarrayNList<TypeVar> "Returns the elements of the possibly nested
mutable ragged array as a list, recursively"
  input MRArray<TypeVar> arr;
  output List<TypeVar> result;
algorithm
 // return the elements of the possibly nested mutable ragged array as a list
end mrarrayNList;

function mrarrayList<TypeVar> "Returns the elements of the mutable ragged array
as a list"
  input MRArray<TypeVar> arr;
  output List<TypeVar> result;
algorithm
 // return the elements of the mutable ragged array as a list
end mrarrayList;

function listNMRArray<TypeVar> "Returns the elements of the possibly nested list
as an MRArray"
  input List<TypeVar> lst;
```

```
  output MRArray<TypeVar> result;
algorithm
 // return the elements of the possibly nested list, recursively as an MRArray
end listNMRArray;

function listMRArray<TypeVar> "Returns the elements of the list as an MRArray"
  input List<TypeVar> lst;
  output MRArray<TypeVar> result;
algorithm
 // return the elements of the list as an MRArray
end listMRArray;

function mrarrayUpdate<TypeVar>
  "Updates the array in place with a new element at
   the given index. The index starts at 1."
  input MRArray<TypeVar> arr;
  input Integer index;
  input TypeVar element;
  output MRArray<TypeVar> result;
algorithm
  arr[index] := element;
end mrarrayUpdate;

function mrarrayCreate<TypeVar>
  "Creates an array of specifed size with all
   the elements intialized with specified element"
  input Integer size;
  input TypeVar element;
  output MRArray<TypeVar> result;
algorithm
  // create an array of specified size with all
  // elements intialized to eleement
end mrarrayCreate;

function mrarrayCopy<TypeVar> "Create a copy of the specifed array"
  input MRArray<TypeVar> arrInput;
  output MRArray<TypeVar> arrOutput;
algorithm
  // create a new array identical with the input and return it as output
end mrarrayCopy;

function mrarrayAdd<TypeVar> "Add an element at the end of the array"
  input MRArray<TypeVar> arrInput;
  input TypeVar element;
  output MRArray<TypeVar> arrOutput;
algorithm
  // create a new array identical with the input add the element
  // at the end and return it as output
end mrarrayAdd;
```

B.3.9 Miscellaneous Operations

The following are a few miscellaneous operations

```
function clock "Returns the current time as a real number"
  output Real output1;
algorithm
  // returns the current time represented as a real number
end clock;

function print "Prints the string given as parameter"
  input String s;
algorithm
  // prints the string s.
end print;

function tick "Returns a unique integer value"
  output Integer i;
```

```
algorithm
  // returns a unique integer value, different from previous
  // values returned by tick during this execution.
end tick;


end MetaModelica;
```

Appendix C

Complete Small Language Specifications

This appendix contains several small language specifications, both interpretive and translational.

C.1 The Complete Interpretive Semantics for PAM

The complete semantics of the PAM language as earlier described in Section 2.5 follows below, but with some minor differences.

```
package Pam
"This version differs from the one in the book. The State is just
 the current environment, the input and output streams do not exist.
 Input is done through the function read which just calls a c function doing
 a call to scanf. Works if no backtracking occurs, as when print is used."

public
type Ident = String;

// Semantics oriented abstract syntax for the PAM language

uniontype BinOp
  record ADD end ADD;
  record SUB end SUB;
  record MUL end MUL;
  record DIV end DIV;
end BinOp;

uniontype RelOp
  record EQ end EQ;
  record GT end GT;
  record LT end LT;
  record LE end LE;
  record GE end GE;
  record NE end NE;
end RelOp;

uniontype Exp
  record INT
    Integer integer;
  end INT;

  record IDENT
    Ident ident;
  end IDENT;

  record BINARY
    Exp exp1;
    BinOp binOp2;
    Exp exp3;
  end BINARY;

  record RELATION
    Exp exp1;
    RelOp relOp2;
    Exp exp3;
  end RELATION;
```

```
end Exp;

type IdentLst = list<Ident>;

uniontype Stmt "statements"

  record ASSIGN "Id := Exp"
    Ident ident;
    Exp id;
  end ASSIGN;

  record IF "if Exp then Stmt ..."
    Exp exp;
    Stmt stmt;
    Stmt if_;
  end IF;

  record WHILE "while Exp do Stmt"
    Exp exp;
    Stmt while_;
  end WHILE;

  record TODO "to Exp do Stmt ..."
    Exp exp;
    Stmt to;
  end TODO;

  record READ "read id1,id2, ..."
    IdentLst read;
  end READ;

  record WRITE "write id1,id2, ..."
    IdentLst write;
  end WRITE;

  record SEQ "Stmt1; Stmt2"
    Stmt stmt;
    Stmt stmt1;
  end SEQ;

  record SKIP "; empty stmt" end SKIP;

end Stmt;

type VarBnd = tuple<Ident,Value>
"Types needed for modeling static and dynamic semantics
 Variable binding and environment type";

type Env = list<VarBnd>;

uniontype Value "Value type needed for evaluation"

  record INTval
    Integer integer;
  end INTval;

  record BOOLval
    Boolean boolean;
  end BOOLval;

end Value;

type State = Env;

protected
```

```
import Input;

function repeatEval "Auxiliary utility functions"
  input State inState;
  input Integer inInteger;
  input Stmt inStmt;
  output State outState;
algorithm
  outstate := matchcontinue (inState,inInteger,inStmt)
    local
      State state,state2,state3;
      Integer n,n2;
      Stmt stmt;

    case (state,n,stmt) "repeatedly evaluate stmt n times n <= 0"
      equation
        (n <= 0) = true;
      then state;

    case (state,n,stmt) "eval n times"
      equation
        (n > 0) = true;
        n2 = n - 1 "n > 0";
        state2 = evalStmt(state, stmt);
        state3 = repeatEval(state2, n2, stmt);
      then state3;

  end matchcontinue;
end repeatEval;

function error
  input String inString1;
  input String inString2;
algorithm
  _ := matchcontinue (inString1,inString2)
    local Ident str1,str2;
    case (str1,str2) "Print error messages str1 and str2, and fail"
      equation
        print("Error - ");
        print(str1);
        print(" ");
        print(str2);
        print("\n");
      then fail();
  end matchcontinue;
end error;

function inputItem
  output Integer i;
algorithm
  i := matchcontinue ()
    /* Disabled due to OpenModelica trying to ceval Input.read()
    case ()
      equation
        print("input: ");
        i = Input.read();
        print("\n");
      then i;
    */
    case () then -1;
  end matchcontinue;
end inputItem;

function outputItem
  input Integer i;
protected
  Ident s;
```

```
algorithm
  s := intString(i);
  print(s);
end outputItem;

function lookup
  input Env inEnv;
  input Ident inIdent;
  output Value outValue;
algorithm
  outValue := matchcontinue (inEnv,inIdent)
    local
      Ident id2,id;
      Value value;
      State rest;

    // lookup returns the value associated with an identifier.
    // If no association is present, lookup will fail. id first in list
    case ((id2,value) :: _,id)
      equation
        equality(id = id2);
      then value;

    case ((id2,_) :: rest,id) "id in rest of list"
      equation
        failure(equality(id = id2));
        value = lookup(rest, id);
      then value;

  end matchcontinue;
end lookup;

function update "update an identifier with a new value in the environment"
  input Env inEnv;
  input Ident inIdent;
  input Value inValue;
  output Env outEnv;
algorithm
  outEnv := matchcontinue (inEnv,inIdent,inValue)
    local
      State env;
      Ident id;
      Value value;
    case (env,id,value) then (id,value) :: env;
  end matchcontinue;
end update;

function applyBinop
  "Apply a binary arithmetic operator to constant integer arguments"
  input BinOp inBinOp1;
  input Integer inInteger2;
  input Integer inInteger3;
  output Integer outInteger;
algorithm
  outInteger := matchcontinue (inBinOp1,inInteger2,inInteger3)
    local Integer x,y;
    case (ADD(),x,y) then x + y;        /* x+y*/
    case (SUB(),x,y) then x - y;        /* x-y */
    case (MUL(),x,y) then x * y;        /* xy */
    case (DIV(),x,y) then intDiv(x, y); /* x/y */
  end matchcontinue;
end applyBinop;

function applyRelop "Apply a function operator, returning a boolean value"
  input RelOp inRelOp1;
  input Integer inInteger2;
  input Integer inInteger3;
```

```
  output Boolean outBoolean;
algorithm
  outBoolean := matchcontinue (inRelOp1,inInteger2,inInteger3)
    local Integer x,y;
    case (LT(),x,y) then (x < y);   // x <  y
    case (LE(),x,y) then (x <= y);  // x <= y
    case (EQ(),x,y) then (x == y);  // x == y
    case (NE(),x,y) then (x <> y);  // x <> y
    case (GE(),x,y) then (x >= y);  // x >= y
    case (GT(),x,y) then (x > y);   // x >  y
  end matchcontinue;
end applyRelop;

function eval "Expression evaluation"
  input Env inEnv;
  input Exp inExp;
  output Value outValue;
algorithm
  outValue := matchcontinue (inEnv,inExp)
    local
      Integer v,v1,v2,v3;
      State env;
      Ident id;
      Exp e1,e2;
      BinOp binop;
      RelOp relop;
      Value val;
      Boolean b;

    case (_,INT(integer = v)) then INTval(v);   // integer constant

    case (env,IDENT(ident = id)) "identifier id"
      equation
        val = lookup(env, id);
      then val;

    // If id not declared, give an error message and
    // fail through error undefined variable id"
    case (env,IDENT(ident = id))
      equation
        failure(_ = lookup(env, id));
        error("Undefined identifier", id);
      then INTval(0);

    case (env,BINARY(exp1 = e1,binOp2 = binop,exp3 = e2)) "expr1 binop expr2"
      equation
        INTval(integer = v1) = eval(env, e1);
        INTval(integer = v2) = eval(env, e2);
        v3 = applyBinop(binop, v1, v2);
      then INTval(v3);

    case (env,RELATION(exp1 = e1,relOp2 = relop,exp3 = e2)) "expr1 relop expr2"
      equation
        INTval(integer = v1) = eval(env, e1);
        INTval(integer = v2) = eval(env, e2);
        b = applyRelop(relop, v1, v2);
      then BOOLval(b);
  end matchcontinue;
end eval;

public
function evalStmt "Statement evaluation"
  input State inState;
  input Stmt inStmt;
  output State outState;
algorithm
  outstate := matchcontinue (inState,inStmt)
```

```
local
  Value v1;
  State env2,env,state2,state1,state,state3;
  Ident id;
  Exp e1,comp;
  Stmt s1,s2,stmt1,stmt2;
  Integer n1,v2;
  IdentLst rest;

// Statement evaluation: map the current state into a new state Assignment
case (env,ASSIGN(ident = id,id = e1))
  equation
    v1 = eval(env, e1);
    env2 = update(env, id, v1);
  then env2;

// IF true ...
case ((state1 as env),IF(exp = comp,stmt = s1,if_ = s2))
  equation
    BOOLval(boolean = true) = eval(env, comp);
    state2 = evalStmt(state1, s1);
  then state2;

// IF false ...
case ((state1 as env),IF(exp = comp,stmt = s1,if_ = s2))
  equation
    BOOLval(boolean = false) = eval(env, comp);
    state2 = evalStmt(state1, s2);
  then state2;

// WHILE ...
case (state,WHILE(exp = comp,while_ = s1))
  equation
    state2 = evalStmt(state, IF(comp,SEQ(s1,WHILE(comp,s1)),SKIP()));
  then state2;

// TO e1 DO s1 ...
case ((state as env),TODO(exp = e1,to = s1))
  equation
    INTval(integer = n1) = eval(env, e1);
    state2 = repeatEval(state, n1, s1);
  then state2;

// READ {}
case (state,READ(read = {})) then state;

// READ id1 ...
case (env,READ(read = id :: rest))
  equation
    v2 = inputItem();
    env2 = update(env, id, INTval(v2));
    state2 = evalStmt(env2, READ(rest));
  then state2;

// WRITE {}
case (state,WRITE(write = {})) then state;

// WRITE id1 ...
case (env,WRITE(write = id :: rest))
  equation
    INTval(integer = v2) = lookup(env, id);
    outputItem(v2);
    state2 = evalStmt(env, WRITE(rest));
  then state2;

// sequence of statements: stmt1 ; stmt2
```

```
    case (state,SEQ(stmt = stmt1,stmt1 = stmt2))
      equation
        state2 = evalStmt(state, stmt1);
        state3 = evalStmt(state2, stmt2);
      then state3;

    // ; empty statement
    case (state,SKIP()) then state;

  end matchcontinue;
end evalStmt;

end Pam;
```

C.2 Complete PAMDECL Interpretive Specification

The complete PAMDECL interpretive specifiation is presented below. The packages/files are shown in the following order, starting with the more interesting semantics packages, ending with the lexer and parsing files.

Main ScanParse Absyn Env Eval lexer.l parser.y Makefile

C.2.1 PAMDECL Main Package

The main package implements the read-eval-print function as the function evalprog, which accepts the initial environment initial containing only true and false exported from package Eval. The main package of the PamDecl evaluator calls ScanParse to read and parse text from the standard input, and Eval to evaluate and print the results.

```
package Main

import ScanParse;
import Eval;
import Absyn;

function main
protected
  Absyn.Prog ast;
algorithm
  print("[Parse. Enter a program, then press ");
  print("CTRL+z (Windows) or CTRL+d (Linux).]\n");
  ast := ScanParse.scanparse();
  Eval.evalprog(ast);
end main;

end Main;
```

C.2.2 PAMDECL ScanParse

The ScanParse package contains only one function scanparse, which is an external function implemented in C to scan and parse text written in the PamDecl language.

```
package ScanParse

public import Absyn;

public function scanparse
  output Absyn.Prog outProg;
external "C" outProg = parse() annotation(Library = {"lexer.o","parser.o"});
end scanparse;

end ScanParse;
```

C.3 PAMDECL Absyn

The Absyn package specifies the abstract syntax used by the rest of the specification, i.e., the other packages.

```
package Absyn

public
type Ident = String;

uniontype BinOp
  record ADD end ADD;
  record SUB end SUB;
  record MUL end MUL;
  record DIV end DIV;
end BinOp;

uniontype UnOp
  record NEG end NEG;
end UnOp;

uniontype RelOp
  record EQ end EQ;
  record GT end GT;
  record LT end LT;
  record LE end LE;
  record GE end GE;
  record NE end NE;
end RelOp;

uniontype Expr

  record INTCONST
    Integer integer;
  end INTCONST;

  record REALCONST
    Real real;
  end REALCONST;

  record BINARY
    Expr expr1;
    BinOp binOp2;
    Expr expr3;
  end BINARY;

  record UNARY
    UnOp unOp;
    Expr expr;
  end UNARY;

  record RELATION
    Expr expr1;
    RelOp relOp2;
    Expr expr3;
  end RELATION;

  record VARIABLE
    Ident ident;
  end VARIABLE;

end Expr;

type StmtLst = list<Stmt>;

uniontype Stmt
```

```
    record ASSIGN
      Ident ident;
      Expr expr;
    end ASSIGN;

    record WRITE
      Expr expr;
    end WRITE;

    record NOOP end NOOP;

    record IF
      Expr expr1;
      StmtLst stmtLst2;
      StmtLst stmtLst3;
    end IF;

    record WHILE
      Expr expr;
      StmtLst stmtLst;
    end WHILE;

  end Stmt;

  type StmtList = list<Stmt>;

  uniontype Decl
    record NAMEDECL
      Ident ident1;
      Ident ident2;
    end NAMEDECL;

  end Decl;

  type DeclList = list<Decl>;

  uniontype Prog

    record PROG
      DeclList declList;
      StmtList stmtList;
    end PROG;

  end Prog;

  end Absyn;
```

C.3.1 PAMDECL Env

The Env package contains functions and types for describing and handling environments in a declarative way, including building environments and performing lookup on environments.

```
  package Env

  public
  type Ident = String;

  public
  uniontype Value

    record INTVAL
      Integer integer;
    end INTVAL;

    record REALVAL
      Real real;
```

```
  end REALVAL;

  record BOOLVAL
    Boolean boolean;
  end BOOLVAL;

end Value;

public
uniontype Value2

  record INTVAL2
    Integer integer1;
    Integer integer2;
  end INTVAL2;

  record REALVAL2
    Real real1;
    Real real2;
  end REALVAL2;

end Value2;

public
uniontype Type
  record INTTYPE end INTTYPE;
  record REALTYPE end REALTYPE;
  record BOOLTYPE end BOOLTYPE;
end Type;

public
uniontype Bind

  record BIND
    Ident ident;
    Type type_;
    Value value;
  end BIND;

end Bind;

public
type Env = list<Bind>;

public function initial_
  output BindLst outBindLst;
  type BindLst = list<Bind>;
algorithm
  outBindLst := matchcontinue ()
    case ()
    then {BIND("false",BOOLTYPE(),BOOLVAL(false)),
          BIND("true",BOOLTYPE(),BOOLVAL(true))};
  end matchcontinue;
end initial_;

public
function lookup
  input Env inEnv;
  input Ident inIdent;
  output Value outValue;
algorithm
  outValue := matchcontinue (inEnv,inIdent)
    local
      Ident idenv,id;
      Value v;
      Env rest;
```

```
    case ((BIND(ident = idenv,value = v) :: _),id)
      equation
        equality(id = idenv);
      then v;

    case ((BIND(ident = idenv) :: rest),id)
      equation
        failure(equality(id = idenv));
        v = lookup(rest, id);
     then v;

  end matchcontinue;
end lookup;

public
function lookuptype
  input Env inEnv;
  input Ident inIdent;
  output Type outType;
algorithm
  outType := matchcontinue (inEnv,inIdent)
    local
      Ident idenv,id;
      Type t;
      Env rest;

    case ((BIND(ident = idenv,type_ = t) :: _),id)
      equation
        equality(id = idenv);
      then t;

    case ((BIND(ident = idenv) :: rest),id)
      equation
        failure(equality(id = idenv));
        t = lookuptype(rest, id);
      then t;

  end matchcontinue;
end lookuptype;

public
function update
  input Env env;
  input Ident id;
  input Type ty;
  input Value v;
  output Env newenv;
algorithm
  newenv := (BIND(id,ty,v) :: env);
end update;

end Env;
```

C.3.2 PAMDECL Eval

The Eval package contains the interpretive semantic cases, collected in functions, for all the constructs in PAMDECL, including expressions, statements, and declarations.

```
package Eval

public
import Absyn;
import Env;

protected
```

```
function binaryLub "Type lattice;  int --> real"
  input Env.Value inValue1;
  input Env.Value inValue2;
  output Env.Value2 outValue2;
algorithm
  outValue2 := matchcontinue (inValue1,inValue2)
    local
      Integer v1,v2;
      Real r1,r2;

    case (Env.INTVAL(integer = v1),Env.INTVAL(integer = v2))
      then Env.INTVAL2(v1,v2);

    case (Env.REALVAL(real = r1),Env.REALVAL(real = r2))
      then Env.REALVAL2(r1,r2);

    case (Env.INTVAL(integer = v1),Env.REALVAL(real = r2))
      equation
        r1 = intReal(v1);
      then Env.REALVAL2(r1,r2);

    case (Env.REALVAL(real = r1),Env.INTVAL(integer = v2))
      equation
        r2 = intReal(v2);
      then Env.REALVAL2(r1,r2);

  end matchcontinue;
end binaryLub;

protected
function promote
  "Promotion and type check"
  input Env.Value inValue;
  input Env.Type inType;
  output Env.Value outValue;
algorithm
  outValue := matchcontinue (inValue,inType)
    local
      Integer v;
      Real r;
      Boolean b;

    case (Env.INTVAL(integer = v),Env.INTTYPE()) then Env.INTVAL(v);
    case (Env.REALVAL(real = r),Env.REALTYPE()) then Env.REALVAL(r);
    case (Env.BOOLVAL(boolean = b),Env.BOOLTYPE()) then Env.BOOLVAL(b);

    // cast integer to real
    case (Env.INTVAL(integer = v),Env.REALTYPE())
      equation
        r = intReal(v);
      then Env.REALVAL(r);

  end matchcontinue;
end promote;

protected
function applyIntBinary
  "Auxiliary functions for applying the binary operators"
  input Absyn.BinOp inBinOp1;
  input Integer inInteger2;
  input Integer inInteger3;
  output Integer outInteger;
algorithm
  outInteger := matchcontinue (inBinOp1,inInteger2,inInteger3)
    local Integer v1,v2;
    case (Absyn.ADD(),v1,v2) then v1 + v2;
    case (Absyn.SUB(),v1,v2) then v1 - v2;
```

```
    case (Absyn.MUL(),v1,v2) then v1 * v2;
    case (Absyn.DIV(),v1,v2) then intDiv(v1,v2);
  end matchcontinue;
end applyIntBinary;

protected
function applyRealBinary
  input Absyn.BinOp inBinOp1;
  input Real inReal2;
  input Real inReal3;
  output Real outReal;
algorithm
  outReal := matchcontinue (inBinOp1,inReal2,inReal3)
    local Real v1,v2;
    case (Absyn.ADD(),v1,v2) then v1 + v2;
    case (Absyn.SUB(),v1,v2) then v1 - v2;
    case (Absyn.MUL(),v1,v2) then v1 * v2;
    case (Absyn.DIV(),v1,v2) then v1 / v2;
  end matchcontinue;
end applyRealBinary;

protected
function applyIntUnary
  "Auxiliary functions for applying the unary operators"
  input Absyn.UnOp inUnOp;
  input Integer inInteger;
  output Integer outInteger;
algorithm
  outInteger := matchcontinue (inUnOp,inInteger)
    local Integer v1;
    case (Absyn.NEG(),v1) then -v1;
  end matchcontinue;
end applyIntUnary;

protected
function applyRealUnary
  input Absyn.UnOp inUnOp;
  input Real inReal;
  output Real outReal;
algorithm
  outReal := matchcontinue (inUnOp,inReal)
    local Real v1;
    case (Absyn.NEG(),v1) then -. v1;
  end matchcontinue;
end applyRealUnary;

protected
function applyIntRelation
  "Auxiliary functions for applying the function operators"
  input Absyn.RelOp inRelOp1;
  input Integer inInteger2;
  input Integer inInteger3;
  output Boolean outBoolean;
algorithm
  outBoolean := matchcontinue (inRelOp1,inInteger2,inInteger3)
    local Integer v1,v2;
    case (Absyn.LT(),v1,v2) then (v1 < v2);
    case (Absyn.LE(),v1,v2) then (v1 <= v2);
    case (Absyn.GT(),v1,v2) then (v1 > v2);
    case (Absyn.GE(),v1,v2) then (v1 >= v2);
    case (Absyn.NE(),v1,v2) then (v1 <> v2);
    case (Absyn.EQ(),v1,v2) then (v1 == v2);
  end matchcontinue;
end applyIntRelation;

protected
```

```
function applyRealRelation
  input Absyn.RelOp inRelOp1;
  input Real inReal2;
  input Real inReal3;
  output Boolean outBoolean;
algorithm
  outBoolean := matchcontinue (inRelOp1,inReal2,inReal3)
    local Real v1,v2;
    case (Absyn.LT(),v1,v2) then (v1 <. v2);
    case (Absyn.LE(),v1,v2) then (v1 <=. v2);
    case (Absyn.GT(),v1,v2) then (v1 >. v2);
    case (Absyn.GE(),v1,v2) then (v1 >=. v2);
    case (Absyn.NE(),v1,v2) then (v1 <>. v2);
    case (Absyn.EQ(),v1,v2) then (v1 ==. v2);
  end matchcontinue;
end applyRealRelation;

protected
function evalExpr
  "EVALUATE A SINGLE EXPRESSION in an environment. Return
   the new value. Expressions do not change environments."
  input Env.Env inEnv;
  input Absyn.Expr inExpr;
  output Env.Value outValue;
algorithm
  outValue := matchcontinue (inEnv,inExpr)
    local
      list<Env.Bind> env;
      Integer v,c1,c2,v3;
      Env.Value v1,v2;
      Absyn.Expr e1,e2;
      Absyn.BinOp binop;
      Absyn.UnOp unop;
      Absyn.RelOp relop;
      String id;
      Real r,r1,r2,r3;
      Boolean b;

    // integer/real constants
    case (env,Absyn.INTCONST(integer = v)) then Env.INTVAL(v);
    case (env,Absyn.REALCONST(real = r)) then Env.REALVAL(r);

    // Binary operators
    case (env,Absyn.BINARY(expr1 = e1,binOp2 = binop,expr3 = e2))
      equation
        v1 = evalExpr(env, e1);
        v2 = evalExpr(env, e2);
        Env.INTVAL2(integer1 = c1,integer2 = c2) = binaryLub(v1, v2);
        v3 = applyIntBinary(binop, c1, c2);
      then Env.INTVAL(v3);

    case (env,Absyn.BINARY(expr1 = e1,binOp2 = binop,expr3 = e2))
      equation
        v1 = evalExpr(env, e1);
        v2 = evalExpr(env, e2);
        Env.REALVAL2(real1 = r1,real2 = r2) = binaryLub(v1, v2);
        r3 = applyRealBinary(binop, r1, r2);
      then Env.REALVAL(r3);

    case (_,Absyn.BINARY(expr1 = _))
      equation
        print("Error: binary operator applied to invalid type(s)\n");
      then fail();

    // Unary operators
    case (env,Absyn.UNARY(unOp = unop,expr = e1))
      equation
```

```
        Env.INTVAL(integer = c1) = evalExpr(env, e1);
        c2 = applyIntUnary(unop, c1);
      then Env.INTVAL(c2);

    case (env,Absyn.UNARY(unOp = unop,expr = e1))
      equation
        Env.REALVAL(real = r1) = evalExpr(env, e1);
        r2 = applyRealUnary(unop, r1);
      then Env.REALVAL(r2);

    case (_,Absyn.UNARY(unOp = _))
      equation
        print("Error: unary operator applied to invalid type\n");
      then fail();

    // Relation operators
    case (env,Absyn.RELATION(expr1 = e1,relOp2 = relop,expr3 = e2))
      equation
        v1 = evalExpr(env, e1);
        v2 = evalExpr(env, e2);
        Env.INTVAL2(integer1 = c1,integer2 = c2) = binaryLub(v1, v2);
        b = applyIntRelation(relop, c1, c2);
      then Env.BOOLVAL(b);

    case (env,Absyn.RELATION(expr1 = e1,relOp2 = relop,expr3 = e2))
      equation
        v1 = evalExpr(env, e1);
        v2 = evalExpr(env, e2);
        Env.REALVAL2(real1 = r1,real2 = r2) = binaryLub(v1, v2);
        b = applyRealRelation(relop, r1, r2);
      then Env.BOOLVAL(b);

    case (_,Absyn.RELATION(expr1 = _))
      equation
        print("Error: relation operator applied to invalid type(s)\n");
      then fail();

    // Variable lookup
    case (env,Absyn.VARIABLE(ident = id))
      equation
        v1 = Env.lookup(env, id);
      then v1;

    case (env,Absyn.VARIABLE(ident = id))
      equation
        failure(_ = Env.lookup(env, id));
        print("Error: undefined variable (");
        print(id);
        print(")\n");
      then fail();

  end matchcontinue;

end evalExpr;

protected
function printValue
  "EVALUATING STATEMENTS
   Print a value - the \"write\" statement"
  input Env.Value inValue;
algorithm
  _ := matchcontinue (inValue)
    local
      String vstr;
      Integer v;
      Real r;
```

```
    case (Env.INTVAL(integer = v))
      equation
        vstr = intString(v);
        print(vstr);
        print("\n");
      then ();

    case Env.REALVAL(real = r)
      equation
        vstr = realString(r);
        print(vstr);
        print("\n");
      then ();

    case Env.BOOLVAL(boolean = true)
      equation
        print("true\n");
      then ();

    case Env.BOOLVAL(boolean = false)
      equation
        print("false\n");
      then ();

  end matchcontinue;
end printValue;

protected
function evalStmt "Evaluate a single statement. Pass environment forward."
  input Env.Env inEnv;
  input Absyn.Stmt inStmt;
  output Env.Env outEnv;
algorithm
  outEnv := matchcontinue (inEnv,inStmt)
    local
      Env.Value v,v2;
      Env.Type ty;
      list<Env.Bind> env1,env,env2;
      String id;
      Absyn.Expr e;
      list<Absyn.Stmt> c,a,ss;

    case (env,Absyn.ASSIGN(ident = id,expr = e))
      equation
        v = evalExpr(env, e);
        ty = Env.lookuptype(env, id);
        v2 = promote(v, ty);
        env1 = Env.update(env, id, ty, v2);
      then env1;

    case (env,Absyn.ASSIGN(ident = id,expr = e))
      equation
        v = evalExpr(env, e);
        print("Error: assignment mismatch or variable missing\n");
      then fail();

    case (env,Absyn.WRITE(expr = e))
      equation
        v = evalExpr(env, e);
        printValue(v);
      then env;

    case (env,Absyn.NOOP()) then env;

    case (env,Absyn.IF(expr1 = e,stmtLst2 = c))
      equation
```

```
        Env.BOOLVAL(boolean = true) = evalExpr(env, e);
        env1 = evalStmtList(env, c);
      then env1;

    case (env,Absyn.IF(expr1 = e,stmtLst3 = a))
      equation
        Env.BOOLVAL(boolean = false) = evalExpr(env, e);
        env1 = evalStmtList(env, a);
      then env1;

    case (env,Absyn.WHILE(expr = e,stmtLst = ss))
      equation
        Env.BOOLVAL(boolean = true) = evalExpr(env, e);
        env1 = evalStmtList(env, ss);
        env2 = evalStmt(env1, Absyn.WHILE(e,ss));
      then env2;

    case (env,Absyn.WHILE(expr = e,stmtLst = ss))
      equation
        Env.BOOLVAL(boolean = false) = evalExpr(env, e);
      then env;

    case (env,Absyn.IF(expr1 = e,stmtLst3 = a))
      equation
        Env.BOOLVAL(boolean = false) = evalExpr(env, e);
        env1 = evalStmtList(env, a);
      then env1;

    case (env,Absyn.WHILE(expr = e,stmtLst = ss))
      equation
        Env.BOOLVAL(boolean = true) = evalExpr(env, e);
        env1 = evalStmtList(env, ss);
        env2 = evalStmt(env1, Absyn.WHILE(e,ss));
      then env2;

    case (env,Absyn.WHILE(expr = e,stmtLst = ss))
      equation
        Env.BOOLVAL(boolean = false) = evalExpr(env, e);
      then env;

  end matchcontinue;
end evalStmt;

protected
function evalStmtList
  "Evaluate a list of statements in an environent.
   Pass environment forward"
  input Env.Env inEnv;
  input Absyn.StmtList inStmtList;
  output Env.Env outEnv;
algorithm
  outEnv := matchcontinue (inEnv,inStmtList)
    local
      list<Env.Bind> env,env1,env2;
      Absyn.Stmt s;
      list<Absyn.Stmt> ss;

    case (env,{}) then env;

    case (env,(s :: ss))
      equation
        env1 = evalStmt(env, s);
        env2 = evalStmtList(env1, ss);
      then env2;

  end matchcontinue;
end evalStmtList;
```

```
protected function evalDecl
"EVALUATING DECLARATIONS
 Evaluate a single statement.
 Pass environment forward."
  input Env.Env inEnv;
  input Absyn.Decl inDecl;
  output Env.Env outEnv;
algorithm
  outEnv := matchcontinue (inEnv,inDecl)
    local
      list<Env.Bind> env2,env;
      String var;

    case (env,Absyn.NAMEDECL(ident1 = var,ident2 = "integer"))
      equation
        env2 = Env.update(env, var, Env.INTTYPE(), Env.INTVAL(0));
      then env2;

    case (env,Absyn.NAMEDECL(ident1 = var,ident2 = "real"))
      equation
        env2 = Env.update(env, var, Env.REALTYPE(), Env.REALVAL(0.0));
      then env2;

    case (env,Absyn.NAMEDECL(ident1 = var,ident2 = "boolean"))
      equation
        env2 = Env.update(env, var, Env.BOOLTYPE(), Env.BOOLVAL(false));
      then env2;

  end matchcontinue;
end evalDecl;

protected
function evalDeclList
"Evaluate a list of declarations, extending the environent."
  input Env.Env inEnv;
  input Absyn.DeclList inDeclList;
  output Env.Env outEnv;
algorithm
  outEnv := matchcontinue (inEnv,inDeclList)
    local
      list<Env.Bind> env,env1,env2;
      Absyn.Decl s;
      list<Absyn.Decl> ss;

    case (env,{}) then env;

    case (env,(s :: ss))
      equation
        env1 = evalDecl(env, s);
        env2 = evalDeclList(env1, ss);
      then env2;
  end matchcontinue;

end evalDeclList;

function evalprog
"EVALUTATING A PROGRAM means to evaluate the list of statements,
 with an initial environment containing just standard defs."
  input Absyn.Prog inProg;
algorithm
  _ := matchcontinue (inProg)
    local
      list<Env.Bind> env1,env2,env3;
      list<Absyn.Decl> decls;
      list<Absyn.Stmt> stmts;
```

```
    case (Absyn.PROG(declList = decls,stmtList = stmts))
      equation
        env1 = Env.initial_();
        env2 = evalDeclList(env1, decls);
        env3 = evalStmtList(env2, stmts);
      then ();

  end matchcontinue;
end evalprog;

end Eval;
```

C.3.3 PAMDECL lexer.l

```
%{
#include <stdlib.h>
#define YYSTYPE void*
#include "parser.h"


#ifdef RML
#include "yacclib.h"
#include "Absyn.h"
#else
#include "meta_modelica.h"
extern struct record_description Absyn_Expr_INTCONST__desc;
extern struct record_description Absyn_Expr_REALCONST__desc;
extern struct record_description Absyn_Expr_BINARY__desc;
extern struct record_description Absyn_Expr_UNARY__desc;
extern struct record_description Absyn_Expr_RELATION__desc;
extern struct record_description Absyn_Expr_VARIABLE__desc;

#define Absyn__INTCONST(X1)     (mmc_mk_box2(3,&Absyn_Expr_INTCONST__desc,X1))
#define Absyn__REALCONST(X1)    (mmc_mk_box2(4,&Absyn_Expr_REALCONST__desc,X1))

#define Absyn__BINARY(X1,OP,X2) (mmc_mk_box4(5,&Absyn_Expr_BINARY__desc,X1,OP,X
2))
#define Absyn__UNARY(OP,X1)     (mmc_mk_box3(6,&Absyn_Expr_UNARY__desc,OP,X1))
#define Absyn__RELATION(X1,OP,X2) (mmc_mk_box4(7,&Absyn_Expr_RELATION__desc,X1,
OP,X2$
#define Absyn__VARIABLE(X1)     (mmc_mk_box2(8,&Absyn_Expr_VARIABLE__desc,X1))
#endif

int absyn_integer(char *s);
int absyn_ident_or_keyword(char *s);
int yywrap();
%}

%option yylineno

%x c_comment

digit           [0-9]
digits          {digit}+
letter          [A-Za-z_]

intcon          {digits}

dot             "."
sign            [+-]
exponent        ([eE]{sign}?{digits})
realcondot      {digits}{dot}{digits}{exponent}?
realconexp      {digits}({dot}{digits})?{exponent}
realcon         {realcondot}|{realconexp}

ident           {letter}({letter}|{digit})*
ws              [ \t\n]
```

```
junk            .|\n

%%

"/\*"       {
               BEGIN(c_comment);
            }
<c_comment>
{
    "\*/"    { BEGIN(INITIAL); }
    "/\*"    { yyerror("Suspicious comment"); }
    [^\n]    ;
    \n       ;
    <<EOF>>  {
               yyerror("Unterminated comment");
               yyterminate();
            }
}

"("             return T_LPAREN;
")"             return T_RPAREN;
"+"             return T_PLUS;
"-"             return T_MINUS;
"*"             return T_TIMES;
"/"             return T_DIVIDE;
":="            return T_ASSIGN;
";"             return T_SEMICOLON;
":"             return T_COLON;
"<"             return T_LT;
"<="            return T_LE;
">"             return T_GT;
">="            return T_GE;
"<>"            return T_NE;
"="             return T_EQ;

{intcon}        { return absyn_integer(yytext);}
{realcon}       { return absyn_real(yytext);}
{ident}         { return absyn_ident_or_keyword(yytext);}

{ws}+           ;
{junk}+         return T_GARBAGE;

%%

/* Make an RML integer from a C string representation (decimal),
   box it for our abstract syntax, put in yylval and return constant token. */

int absyn_integer(char *s)
{
  yylval = Absyn__INTCONST(mmc_mk_icon(atoi(s)));
  return T_CONST_INT;
}

/* Make an RML real from a C string representation,
   box it for our abstract syntax, put in yylval and return constant token. */

int absyn_real(char *s)
{
  yylval = Absyn__REALCONST(mmc_mk_rcon(atof(s)));
  return T_CONST_REAL;
}

/* Make an RML Ident or a keyword token from a C string */

static struct keyword_s
{
  char *name;
```

```
  int token;
} kw[] =
{
  {"body",      T_BODY},
  {"do",        T_DO},
  {"else",      T_ELSE},
  {"end",       T_END},
  {"if",        T_IF},
  {"program",   T_PROGRAM},
  {"then",      T_THEN},
  {"while",     T_WHILE},
  {"write",     T_WRITE},
};

int absyn_ident_or_keyword(char *s)
{
  int low = 0;
  int high = (sizeof kw) / sizeof(struct keyword_s) - 1;

  while( low <= high ) {
    int mid = (low + high) / 2;
    int cmp = strcmp(kw[mid].name, yytext);
    if( cmp == 0 )
      return kw[mid].token;
    else if( cmp < 0 )
      low = mid + 1;
    else
      high = mid - 1;
  }
  yylval = mmc_mk_scon(s);
  return T_IDENT;
}

int yywrap()
{
        return 1;
}
```

C.3.4 PAMDECL parser.y

```
%{
#include <stdio.h>

void yyerror(char *str);
typedef void *rml_t;
#define YYSTYPE rml_t
rml_t absyntree;

void* parse()
{
  yyparse();
  return absyntree;
}

#ifdef RML
#include "yacclib.h"
#include "Absyn.h"
#else
#include "meta_modelica.h"

/* Namedecl */
extern struct record_description Absyn_Decl_NAMEDECL__desc;

#define Absyn__NAMEDECL(X1,X2) (mmc_mk_box3(3,&Absyn_Decl_NAMEDECL__desc,X1,X2)
)

/* Program */
```

```
extern struct record_description Absyn_Prog_PROG__desc;

#define Absyn__PROG(X1,X2)      (mmc_mk_box3(3,&Absyn_Prog_PROG__desc,X1,X2))

/* BinOp */
extern struct record_description Absyn_BinOp_ADD__desc;
extern struct record_description Absyn_BinOp_SUB__desc;
extern struct record_description Absyn_BinOp_MUL__desc;
extern struct record_description Absyn_BinOp_DIV__desc;

#define Absyn__ADD (mmc_mk_box1(3,&Absyn_BinOp_ADD__desc))
#define Absyn__SUB (mmc_mk_box1(4,&Absyn_BinOp_SUB__desc))
#define Absyn__MUL (mmc_mk_box1(5,&Absyn_BinOp_MUL__desc))
#define Absyn__DIV (mmc_mk_box1(6,&Absyn_BinOp_DIV__desc))

/* RelOp */
extern struct record_description Absyn_RelOp_EQ__desc;
extern struct record_description Absyn_RelOp_GT__desc;
extern struct record_description Absyn_RelOp_LT__desc;
extern struct record_description Absyn_RelOp_LE__desc;
extern struct record_description Absyn_RelOp_GE__desc;
extern struct record_description Absyn_RelOp_NE__desc;

#define Absyn__EQ (mmc_mk_box1(3,&Absyn_RelOp_EQ__desc))
#define Absyn__GT (mmc_mk_box1(4,&Absyn_RelOp_GT__desc))
#define Absyn__LT (mmc_mk_box1(5,&Absyn_RelOp_LT__desc))
#define Absyn__LE (mmc_mk_box1(6,&Absyn_RelOp_LE__desc))
#define Absyn__GE (mmc_mk_box1(7,&Absyn_RelOp_GE__desc))
#define Absyn__NE (mmc_mk_box1(8,&Absyn_RelOp_NE__desc))

/* Expr */
extern struct record_description Absyn_Expr_INTCONST__desc;
extern struct record_description Absyn_Expr_REALCONST__desc;
extern struct record_description Absyn_Expr_BINARY__desc;
extern struct record_description Absyn_Expr_UNARY__desc;
extern struct record_description Absyn_Expr_RELATION__desc;
extern struct record_description Absyn_Expr_VARIABLE__desc;

#define Absyn__INTCONST(X1)     (mmc_mk_box2(3,&Absyn_Expr_INTCONST__desc,X1))
#define Absyn__REALCONST(X1)    (mmc_mk_box2(4,&Absyn_Expr_REALCONST__desc,X1))

#define Absyn__BINARY(X1,OP,X2) (mmc_mk_box4(5,&Absyn_Expr_BINARY__desc,X1,OP,X
2))
#define Absyn__UNARY(OP,X1)     (mmc_mk_box3(6,&Absyn_Expr_UNARY__desc,OP,X1))
#define Absyn__RELATION(X1,OP,X2) (mmc_mk_box4(7,&Absyn_Expr_RELATION__desc,X1,
OP,X2))
#define Absyn__VARIABLE(X1)     (mmc_mk_box2(8,&Absyn_Expr_VARIABLE__desc,X1))

/* Stmt */
extern struct record_description Absyn_Stmt_ASSIGN__desc;
extern struct record_description Absyn_Stmt_WRITE__desc;
extern struct record_description Absyn_Stmt_NOOP__desc;
extern struct record_description Absyn_Stmt_IF__desc;
extern struct record_description Absyn_Stmt_WHILE__desc;
extern struct record_description Absyn_Stmt_VARIABLE__desc;

#define Absyn__ASSIGN(X1,X2) (mmc_mk_box3(3,&Absyn_Stmt_ASSIGN__desc,X1,X2))
#define Absyn__WRITE(X1)     (mmc_mk_box2(4,&Absyn_Stmt_WRITE__desc,X1))
#define Absyn__NOOP          (mmc_mk_box1(5,&Absyn_Stmt_NOOP__desc))
#define Absyn__IF(X1,X2,X3)  (mmc_mk_box4(6,&Absyn_Stmt_IF__desc,X1,X2,X3))
#define Absyn__WHILE(X1,X2)  (mmc_mk_box3(7,&Absyn_Stmt_WHILE__desc,X1,X2))

#endif
%}

%token T_PROGRAM
%token T_BODY
```

```
%token T_END
%token T_IF
%token T_THEN
%token T_ELSE
%token T_WHILE
%token T_DO

%token T_WRITE
%token T_ASSIGN
%token T_SEMICOLON
%token T_COLON

%token T_CONST_INT
%token T_CONST_REAL
%token T_CONST_BOOL
%token T_IDENT

%token T_LPAREN T_RPAREN

%nonassoc T_LT T_LE T_GT T_GE T_NE T_EQ
%left T_PLUS  T_MINUS
%left T_TIMES T_DIVIDE
%left T_UMINUS

%token T_GARBAGE

%%

program
        : T_PROGRAM decl_list T_BODY stmt_list T_END T_PROGRAM
        { absyntree = Absyn__PROG($2,$4); }

decl_list
        :
            { $$ = mmc_mk_nil();}
        | decl decl_list
            { $$ = mmc_mk_cons($1,$2); }

decl
        : T_IDENT T_COLON T_IDENT T_SEMICOLON
        { $$ = Absyn__NAMEDECL($1,$3);}

stmt_list
        :
            { $$ = mmc_mk_nil();}
        | stmt stmt_list
            { $$ = mmc_mk_cons($1,$2); }

stmt
        : simple_stmt T_SEMICOLON
        | combined_stmt

simple_stmt
        : assign_stmt
        | write_stmt
        | noop_stmt

combined_stmt
        : if_stmt
        | while_stmt

assign_stmt
: T_IDENT T_ASSIGN expr
            { $$ = Absyn__ASSIGN($1,$3);}

write_stmt
        : T_WRITE expr
```

```
            { $$ = Absyn__WRITE($2);}

noop_stmt
        :
            { $$ = Absyn__NOOP;}

if_stmt
        : T_IF expr T_THEN stmt_list T_ELSE stmt_list T_END T_IF
            { $$ = Absyn__IF($2,$4,$6); }
        | T_IF expr T_THEN stmt_list T_END T_IF
            { $$ = Absyn__IF($2,$4,mmc_mk_cons(Absyn__NOOP,mmc_mk_nil())); }

while_stmt
        : T_WHILE expr T_DO stmt_list T_END T_WHILE
            { $$ = Absyn__WHILE($2,$4); }

expr
        : T_CONST_INT
        | T_CONST_REAL
        | T_CONST_BOOL
        | T_LPAREN expr T_RPAREN
            { $$ = $2;}
        | T_IDENT
            { $$ = Absyn__VARIABLE($1);}
        | expr_bin
        | expr_un
        | expr_rel

expr_bin
        : expr T_PLUS expr
            { $$ = Absyn__BINARY($1, Absyn__ADD,$3);}
        | expr T_MINUS expr
            { $$ = Absyn__BINARY($1, Absyn__SUB,$3);}
        | expr T_TIMES expr
            { $$ = Absyn__BINARY($1, Absyn__MUL,$3);}
        | expr T_DIVIDE expr
            { $$ = Absyn__BINARY($1, Absyn__DIV,$3);}

expr_un
        : T_MINUS expr %prec T_UMINUS
            { $$ = Absyn__UNARY(Absyn__ADD,$2);}

expr_rel
        : expr T_LT expr
            { $$ = Absyn__RELATION($1,Absyn__LT,$3);}
        | expr T_LE expr
            { $$ = Absyn__RELATION($1,Absyn__LE,$3);}
        | expr T_GT expr
            { $$ = Absyn__RELATION($1,Absyn__GT,$3);}
        | expr T_GE expr
            { $$ = Absyn__RELATION($1,Absyn__GE,$3);}
        | expr T_NE expr
            { $$ = Absyn__RELATION($1,Absyn__NE,$3);}
        | expr T_EQ expr
            { $$ = Absyn__RELATION($1,Absyn__EQ,$3);}

%%

void yyerror(char *str) {
        printf("%s on line %d!\n", str, -1);
}
```

C.3.5 PAMDECL Makefile

```
SOLUTIONS=SCRIPT.mos
CLEAN=Absyn_* Env_* Eval_* Main_* Pam_* ScanParse_*
```

```
DEPS=lexer.o parser.o
include ../common.omc
```

C.4 The Complete PAM Translational Specification

The following files are needed for building the PAM translator: Absyn.mo, Trans.mo, Mcode.mo, Emit.mo, lexer.l, parser.y, Main.mo, Parse.mo and Makefile.

C.4.1 PAMTRANS Absyn.mo

```
package Absyn "Semantics oriented abstract syntax for the PAM language"

public
type Ident = String;

public
uniontype BinOp
  record ADD end ADD;
  record SUB end SUB;
  record MUL end MUL;
  record DIV end DIV;
end BinOp;

uniontype RelOp
  record EQ end EQ;
  record GT end GT;
  record LT end LT;
  record LE end LE;
  record GE end GE;
  record NE end NE;
end RelOp;

uniontype Exp

  record INT
    Integer integer;
  end INT;

  record IDENT
    Ident ident;
  end IDENT;

  record BINARY
    Exp exp1;
    BinOp binOp2;
    Exp exp3;
  end BINARY;

  record RELATION
    Exp exp1;
    RelOp relOp2;
    Exp exp3;
  end RELATION;

end Exp;

type Comparison = Exp;

type IdentLst = list<Ident>;

uniontype Stmt

  record ASSIGN "Id := Exp"
    Ident ident;
    Exp id;
  end ASSIGN;
```

```
    record IF  "if Exp then Stmt.."
      Exp exp;
      Stmt stmt;
      Stmt if_;
    end IF;

    record WHILE "while Exp do Stmt"
      Exp exp;
      Stmt while_;
    end WHILE;

    record TODO "to Exp do Stmt..."
      Exp exp;
      Stmt to;
    end TODO;

    record READ "read id1,id2,..."
      IdentLst read;
    end READ;

    record WRITE "write id1,id2,.."
      IdentLst write;
    end WRITE;

    record SEQ "Stmt1; Stmt2"
      Stmt stmt;
      Stmt stmt1;
    end SEQ;

    record SKIP "; empty stmt" end SKIP;

  end Stmt;
  end Absyn;
```

C.4.2 PAMTRANS Trans.mo

```
  package Trans

  public
  import Absyn;
  import Mcode;

  protected
  function transExpr "Arithmetic expression translation
    Evaluation of expressions in the current environment"
    input Absyn.Exp inExp;
    output Mcode_MCodeLst outMcodeMCodeLst;
    type Mcode_MCodeLst = list<Mcode.MCode>;
  algorithm
    outMcodeMCodeLst := matchcontinue (inExp)
      local
        Integer v;
        String id;
        Mcode_MCodeLst cod1,cod3,cod2;
        Mcode.MOperand operand2,t1,t2;
        Mcode.MBinOp opcode;
        Absyn.Exp e1,e2;
        Absyn.BinOp binop;

      // integer constant
      case (Absyn.INT(integer = v)) then {Mcode.MLOAD(Mcode.N(v))};

      // identifier id
      case (Absyn.IDENT(ident = id)) then {Mcode.MLOAD(Mcode.I(id))};

      // Arith binop: simple case, expr2
```

```
    // is just an identifier or constant expr1 binop expr2
    case (Absyn.BINARY(exp1 = e1,binOp2 = binop,exp3 = e2))
      equation
        cod1 = transExpr(e1);
        {Mcode.MLOAD(mOperand = operand2)} = transExpr(e2);
        opcode = transBinop(binop) "expr2 simple";
        cod3 = listAppend(cod1, {Mcode.MB(opcode,operand2)});
      then cod3;

    // Arith binop: general case, expr2
    // is a more complicated expr expr1 binop expr2
    case (Absyn.BINARY(exp1 = e1,binOp2 = binop,exp3 = e2))
      equation
        cod1 = transExpr(e1);
        cod2 = transExpr(e2);
        opcode = transBinop(binop);
        t1 = gentemp();
        t2 = gentemp();
        // code for expr1
        // store expr1
        // code for expr2
        // store expr2
        // load expr1 value into Acc
        // do arith operation
        cod3 = listAppend6(cod1, {Mcode.MSTO(t1)}, cod2, {Mcode.MSTO(t2)},
          {Mcode.MLOAD(t1)}, {Mcode.MB(opcode,t2)});
      then cod3;

  end matchcontinue;
end transExpr;

function transBinop
  input Absyn.BinOp inBinOp;
  output Mcode.MBinOp outMBinOp;
algorithm
  outMBinOp := matchcontinue (inBinOp)
    case (Absyn.ADD()) then Mcode.MADD();
    case (Absyn.SUB()) then Mcode.MSUB();
    case (Absyn.MUL()) then Mcode.MMULT();
    case (Absyn.DIV()) then Mcode.MDIV();
  end matchcontinue;
end transBinop;

function gentemp "generate a temporary"
  output Mcode.MOperand outMOperand;
protected
  Integer no;
algorithm
  no := tick();
  outMOperand := Mcode.T(no);
end gentemp;

function genlabel
  output Mcode.MOperand outMOperand;
protected
  Integer no;
algorithm
  no := tick();
  outMOperand := Mcode.L(no);
end genlabel;

function listAppend3<Type_a>
  input VType_aLst l1;
  input VType_aLst l2;
  input VType_aLst l3;
  output VType_aLst l13;
```

```
public
  type VType_aLst = list<Type_a>;
protected
  VType_aLst l12;
algorithm
  l12 := listAppend(l1, l2);
  l13 := listAppend(l12, l3);
end listAppend3;

function listAppend5<Type_a>
  input VType_aLst l1;
  input VType_aLst l2;
  input VType_aLst l3;
  input VType_aLst l4;
  input VType_aLst l5;
  output VType_aLst l15;
public
  type VType_aLst = list<Type_a>;
protected
  VType_aLst l13;
algorithm
  l13 := listAppend3(l1, l2, l3);
  l15 := listAppend3(l13, l4, l5);
end listAppend5;

function listAppend6<Type_a>
  input VType_aLst l1;
  input VType_aLst l2;
  input VType_aLst l3;
  input VType_aLst l4;
  input VType_aLst l5;
  input VType_aLst l6;
  output VType_aLst l16;
public
  type VType_aLst = list<Type_a>;
protected
  VType_aLst l13,l46;
algorithm
  l13 := listAppend3(l1, l2, l3);
  l46 := listAppend3(l4, l5, l6);
  l16 := listAppend(l13, l46);
end listAppend6;

function listAppend10<Type_a>
  input VType_aLst l1;
  input VType_aLst l2;
  input VType_aLst l3;
  input VType_aLst l4;
  input VType_aLst l5;
  input VType_aLst l6;
  input VType_aLst l7;
  input VType_aLst l8;
  input VType_aLst l9;
  input VType_aLst l10;
  output VType_aLst l110;
public
  type VType_aLst = list<Type_a>;
protected
  VType_aLst l15;
algorithm
  l15 := listAppend5(l1, l2, l3, l4, l5);
  l110 := listAppend6(l15, l6, l7, l8, l9, l10);
end listAppend10;

function transComparison
  input Absyn.Comparison inComparison;
  input Mcode.MOperand inMOperand;
```

```
  output Mcode_MCodeLst outMcodeMCodeLst;
  type Mcode_MCodeLst = list<Mcode.MCode>;
algorithm
  outMcodeMCodeLst := matchcontinue (inComparison,inMOperand)
    local
      Mcode_MCodeLst cod1,cod3,cod2;
      Mcode.MOperand operand2,lab,t1;
      Mcode.MCondJmp jmpop;
      Absyn.Exp e1,e2;
      Absyn.RelOp relop;

    // translation of a comparison:  expr1 function expr2
    // Example call:  trans_comparisonRELATIONINDENTx), GT, INT5)), L10))
    //  Use a simple code pattern the first rule), when expr2 is  a simple
    //  identifier or constant:
    //    code for expr1
    //    SUB operand2
    //    conditional jump to lab
    //  or a general code pattern second rule), which is needed when expr2
    //  is more complicated than a simple identifier or constant:
    //    code for expr1
    //    STO temp1
    //    code for expr2
    //    SUB temp1
    //    conditional jump to lab
    // expr1 relop expr2
    case (Absyn.RELATION(exp1 = e1,relOp2 = relop,exp3 = e2),lab)
      equation
        cod1 = transExpr(e1);
        {Mcode.MLOAD(mOperand = operand2)} = transExpr(e2);
        jmpop = transRelop(relop);
        cod3 = listAppend3(cod1, {Mcode.MB(Mcode.MSUB(),operand2)},
          {Mcode.MJ(jmpop,lab)});
      then cod3;

    // expr1 relop expr2
    case (Absyn.RELATION(exp1 = e1,relOp2 = relop,exp3 = e2),lab)
      equation
        cod1 = transExpr(e1);
        cod2 = transExpr(e2);
        jmpop = transRelop(relop);
        t1 = gentemp();
        cod3 = listAppend5(cod1, {Mcode.MSTO(t1)}, cod2,
          {Mcode.MB(Mcode.MSUB(),t1)},
          {Mcode.MJ(jmpop,lab)});
      then cod3;

  end matchcontinue;
end transComparison;

protected function transRelop
  input Absyn.RelOp inRelOp;
  output Mcode.MCondJmp outMCondJmp;
algorithm
  outMCondJmp := matchcontinue (inRelOp)
    case (Absyn.EQ()) then Mcode.MJNP(); /* Jump on Negative or Positive */
    case (Absyn.LE()) then Mcode.MJP();  /* Jump on Positive */
    case (Absyn.LT()) then Mcode.MJPZ(); /* Jump on Positive or Zero */
    case (Absyn.GT()) then Mcode.MJNZ(); /* Jump on Negative or Zero */
    case (Absyn.GE()) then Mcode.MJN();  /* Jump on Negative */
    case (Absyn.NE()) then Mcode.MJZ();  /* Jump on Zero */
  end matchcontinue;
end transRelop;

protected function transStmt "Statement translation"
  input Absyn.Stmt inStmt;
  output Mcode_MCodeLst outMcodeMCodeLst;
```

```
  type Mcode_MCodeLst = list<Mcode.MCode>;
algorithm
  outMcodeMCodeLst := matchcontinue (inStmt)
    local
      Mcode_MCodeLst cod1,cod2,s1cod,compcod,cod3,s2cod,bodycod,tocod;
      String id;
      Absyn.Exp e1,comp;
      Mcode.MOperand l1,l2,t1;
      Absyn.Stmt s1,s2,stmt1,stmt2;
      list<String> idlist_rest;

    // Statement translation: map the current state into a new state correct??
    // Assignment
    case (Absyn.ASSIGN(ident = id,id = e1))
      equation
        cod1 = transExpr(e1);
        cod2 = listAppend(cod1, {Mcode.MSTO(Mcode.I(id))});
      then cod2;

    // empty statement
    case (Absyn.SKIP()) then {};

    // IF comp then s1
    case (Absyn.IF(exp = comp,stmt = s1,if_ = Absyn.SKIP()))
      equation
        s1cod = transStmt(s1);
        l1 = genlabel();
        compcod = transComparison(comp, l1);
        cod3 = listAppend3(compcod, s1cod, {Mcode.MLABEL(l1)});
      then cod3;

    // IF comp then s1 else s2
    case (Absyn.IF(exp = comp,stmt = s1,if_ = s2))
      equation
        s1cod = transStmt(s1);
        s2cod = transStmt(s2);
        l1 = genlabel();
        l2 = genlabel();
        compcod = transComparison(comp, l1);
        cod3 = listAppend6(compcod, s1cod,
          {Mcode.MJMP(l2)}, {Mcode.MLABEL(l1)},
          s2cod, {Mcode.MLABEL(l2)});
      then cod3;

    // WHILE ...
    case (Absyn.WHILE(exp = comp,while_ = s1))
      equation
        bodycod = transStmt(s1);
        l1 = genlabel();
        l2 = genlabel();
        compcod = transComparison(comp, l2);
        cod3 = listAppend5({Mcode.MLABEL(l1)}, compcod, bodycod,
          {Mcode.MJMP(l1)}, {Mcode.MLABEL(l2)});
      then cod3;

    // TO e1 DO s1 ..
    case (Absyn.TODO(exp = e1,to = s1))
      equation
        tocod = transExpr(e1);
        bodycod = transStmt(s1);
        t1 = gentemp();
        l1 = genlabel();
        l2 = genlabel();
        cod3 = listAppend10(tocod, {Mcode.MSTO(t1)}, {Mcode.MLABEL(l1)},
          {Mcode.MLOAD(t1)}, {Mcode.MB(Mcode.MSUB(),Mcode.N(1))},
          {Mcode.MJ(Mcode.MJN(),l2)},
          {Mcode.MSTO(t1)}, bodycod, {Mcode.MJMP(l1)},
```

```
            {Mcode.MLABEL(l2)});
        then cod3;

      // READ {}
      case (Absyn.READ(read = {})) then {};
      case (Absyn.READ(read = id :: idlist_rest))
        equation
          cod2 = transStmt(Absyn.READ(idlist_rest));
        then Mcode.MGET(Mcode.I(id)) :: cod2;

      // WRITE {}
      case (Absyn.WRITE(write = {})) then {};
      case (Absyn.WRITE(write = id :: idlist_rest))
        equation
          cod2 = transStmt(Absyn.WRITE(idlist_rest));
        then Mcode.MPUT(Mcode.I(id)) :: cod2;

      // sequence of statements: stmt1 ; stmt2
      case (Absyn.SEQ(stmt = stmt1,stmt1 = stmt2))
        equation
          cod1 = transStmt(stmt1);
          cod2 = transStmt(stmt2);
          cod3 = listAppend(cod1, cod2);
        then cod3;

    end matchcontinue;
  end transStmt;

  public function transProgram
    input Absyn.Stmt progbody;
    output Mcode_MCodeLst programcode;
    type Mcode_MCodeLst = list<Mcode.MCode>;
  protected
    Mcode_MCodeLst cod1;
  algorithm
    cod1 := transStmt(progbody);
    programcode := listAppend(cod1, {Mcode.MHALT()});
  end transProgram;

  end Trans;
```

C.4.3 PAMTRANS Mcode.mo

```
  package Mcode

  type Id = String;

  uniontype MBinOp
    record MADD end MADD;

    record MSUB end MSUB;

    record MMULT end MMULT;

    record MDIV end MDIV;

  end MBinOp;

  uniontype MCondJmp
    record MJNP end MJNP;

    record MJP end MJP;

    record MJN end MJN;

    record MJNZ end MJNZ;
```

```
    record MJPZ end MJPZ;

    record MJZ end MJZ;

  end MCondJmp;

  uniontype MOperand
    record I
      Id id;
    end I;

    record N
      Integer integer;
    end N;

    record T
      Integer integer;
    end T;

    record L
      Integer datatype "datatype MLab =  L of int
                        type MTemp    =  T of int
                        type MIdent   =  I of Id
                        type MIdTemp  =  I of Id | T of int";
    end L;

  end MOperand;

  uniontype MCode "datatype MLab =  L of int
                   type MTemp    =  T of int
                   type MIdent   =  I of Id
                   type MIdTemp  =  I of Id | T of int"
    record MB
      MBinOp mBinOp;
      MOperand binary "Binary arith ops";
    end MB;

    record MJ
      MCondJmp mCondJmp;
      MOperand conditional "Conditional jumps";
    end MJ;

    record MJMP
      MOperand mOperand;
    end MJMP;

    record MLOAD
      MOperand mOperand;
    end MLOAD;

    record MSTO
      MOperand mOperand;
    end MSTO;

    record MGET
      MOperand mOperand;
    end MGET;

    record MPUT
      MOperand mOperand;
    end MPUT;

    record MLABEL
      MOperand mOperand;
    end MLABEL;

    record MHALT end MHALT;
```

```
end MCode;
end Mcode;
```

C.4.4 PAMTRANS Emit.mo

```
encapsulated package Emit "automatically generated code from template"

public import Tpl;

public import Mcode;

protected function lm_1
  input Tpl.Text in_txt;
  input list<Mcode.MCode> in_items;

  output Tpl.Text out_txt;
algorithm
  out_txt :=
  matchcontinue(in_txt, in_items)
    local
      Tpl.Text txt;
      list<Mcode.MCode> rest;
      Mcode.MCode i_instruction;

    case ( txt,
           {} )
      then txt;

    case ( txt,
           i_instruction :: rest )
      equation
        txt = emitInstr(txt, i_instruction);
        txt = lm_1(txt, rest);
      then txt;

    case ( txt,
           _ :: rest )
      equation
        txt = lm_1(txt, rest);
      then txt;
  end matchcontinue;
end lm_1;

public function emitAssembly
  input Tpl.Text txt;
  input list<Mcode.MCode> a_lst;

  output Tpl.Text out_txt;
algorithm
  out_txt := lm_1(txt, a_lst);
end emitAssembly;

public function emitInstr
  input Tpl.Text in_txt;
  input Mcode.MCode in_a_code;

  output Tpl.Text out_txt;
algorithm
  out_txt :=
  matchcontinue(in_txt, in_a_code)
    local
      Tpl.Text txt;
      Mcode.MOperand i_mOperand;
      Mcode.MOperand i_conditional;
      Mcode.MCondJmp i_mCondJmp;
      Mcode.MOperand i_binary;
```

```
  Mcode.MBinOp i_mBinOp;
  Tpl.Text txt_1;
  Tpl.Text txt_0;

case ( txt,
       Mcode.MB(mBinOp = i_mBinOp, binary = i_binary) )
  equation
    txt_0 = mbinopToStr(Tpl.emptyTxt, i_mBinOp);
    txt = emitOpOperand(txt, Tpl.textString(txt_0), i_binary);
  then txt;

case ( txt,
       Mcode.MJ(mCondJmp = i_mCondJmp, conditional = i_conditional) )
  equation
    txt_1 = mjmpopToStr(Tpl.emptyTxt, i_mCondJmp);
    txt = emitOpOperand(txt, Tpl.textString(txt_1), i_conditional);
  then txt;

case ( txt,
       Mcode.MJMP(mOperand = i_mOperand) )
  equation
    txt = emitOpOperand(txt, "J", i_mOperand);
  then txt;

case ( txt,
       Mcode.MLOAD(mOperand = i_mOperand) )
  equation
    txt = emitOpOperand(txt, "LOAD", i_mOperand);
  then txt;

case ( txt,
       Mcode.MSTO(mOperand = i_mOperand) )
  equation
    txt = emitOpOperand(txt, "STO", i_mOperand);
  then txt;

case ( txt,
       Mcode.MGET(mOperand = i_mOperand) )
  equation
    txt = emitOpOperand(txt, "GET", i_mOperand);
  then txt;

case ( txt,
       Mcode.MPUT(mOperand = i_mOperand) )
  equation
    txt = emitOpOperand(txt, "PUT", i_mOperand);
  then txt;

case ( txt,
       Mcode.MLABEL(mOperand = i_mOperand) )
  equation
    txt = emitMoperand(txt, i_mOperand);
    txt = Tpl.writeTok(txt, Tpl.ST_STRING("\t"));
    txt = Tpl.writeTok(txt, Tpl.ST_STRING("LAB"));
    txt = Tpl.writeTok(txt, Tpl.ST_NEW_LINE());
  then txt;

case ( txt,
       Mcode.MHALT() )
  equation
    txt = Tpl.writeTok(txt, Tpl.ST_STRING("\t"));
    txt = Tpl.writeTok(txt, Tpl.ST_STRING("HALT"));
    txt = Tpl.writeTok(txt, Tpl.ST_NEW_LINE());
  then txt;

case ( txt,
       _ )
```

```
      then txt;
  end matchcontinue;
end emitInstr;

public function emitOpOperand
  input Tpl.Text txt;
  input String a_opstr;
  input Mcode.MOperand a_op;

  output Tpl.Text out_txt;
algorithm
  out_txt := Tpl.writeTok(txt, Tpl.ST_STRING("\t"));
  out_txt := Tpl.writeStr(out_txt, a_opstr);
  out_txt := Tpl.writeTok(out_txt, Tpl.ST_STRING("\t"));
  out_txt := emitMoperand(out_txt, a_op);
  out_txt := Tpl.writeTok(out_txt, Tpl.ST_NEW_LINE());
end emitOpOperand;

public function emitMoperand
  input Tpl.Text in_txt;
  input Mcode.MOperand in_a_op;

  output Tpl.Text out_txt;
algorithm
  out_txt :=
  matchcontinue(in_txt, in_a_op)
    local
      Tpl.Text txt;
      Integer i_datatype;
      Integer i_integer;
      Mcode.Id i_id;

    case ( txt,
           Mcode.I(id = i_id) )
      equation
        txt = Tpl.writeStr(txt, i_id);
      then txt;

    case ( txt,
           Mcode.N(integer = i_integer) )
      equation
        txt = Tpl.writeStr(txt, intString(i_integer));
      then txt;

    case ( txt,
           Mcode.L(datatype = i_datatype) )
      equation
        txt = Tpl.writeTok(txt, Tpl.ST_STRING("L"));
        txt = Tpl.writeStr(txt, intString(i_datatype));
      then txt;

    case ( txt,
           Mcode.T(integer = i_integer) )
      equation
        txt = Tpl.writeTok(txt, Tpl.ST_STRING("T"));
        txt = Tpl.writeStr(txt, intString(i_integer));
      then txt;

    case ( txt,
           _ )
      then txt;
  end matchcontinue;
end emitMoperand;

public function mbinopToStr
  input Tpl.Text in_txt;
  input Mcode.MBinOp in_a_op;
```

```
  output Tpl.Text out_txt;
algorithm
  out_txt :=
  matchcontinue(in_txt, in_a_op)
    local
      Tpl.Text txt;

    case ( txt,
           Mcode.MADD() )
      equation
        txt = Tpl.writeTok(txt, Tpl.ST_STRING("ADD"));
      then txt;

    case ( txt,
           Mcode.MSUB() )
      equation
        txt = Tpl.writeTok(txt, Tpl.ST_STRING("SUB"));
      then txt;

    case ( txt,
           Mcode.MMULT() )
      equation
        txt = Tpl.writeTok(txt, Tpl.ST_STRING("MULT"));
      then txt;

    case ( txt,
           Mcode.MDIV() )
      equation
        txt = Tpl.writeTok(txt, Tpl.ST_STRING("DIV"));
      then txt;

    case ( txt,
           _ )
      then txt;
  end matchcontinue;
end mbinopToStr;

public function mjmpopToStr
  input Tpl.Text in_txt;
  input Mcode.MCondJmp in_a_jmp;

  output Tpl.Text out_txt;
algorithm
  out_txt :=
  matchcontinue(in_txt, in_a_jmp)
    local
      Tpl.Text txt;

    case ( txt,
           Mcode.MJNP() )
      equation
        txt = Tpl.writeTok(txt, Tpl.ST_STRING("JNP"));
      then txt;

    case ( txt,
           Mcode.MJP() )
      equation
        txt = Tpl.writeTok(txt, Tpl.ST_STRING("JP"));
      then txt;

    case ( txt,
           Mcode.MJN() )
      equation
        txt = Tpl.writeTok(txt, Tpl.ST_STRING("JN"));
      then txt;
```

```
    case ( txt,
           Mcode.MJNZ() )
      equation
        txt = Tpl.writeTok(txt, Tpl.ST_STRING("JNZ"));
      then txt;

    case ( txt,
           Mcode.MJPZ() )
      equation
        txt = Tpl.writeTok(txt, Tpl.ST_STRING("JPZ"));
      then txt;

    case ( txt,
           Mcode.MJZ() )
      equation
        txt = Tpl.writeTok(txt, Tpl.ST_STRING("JZ"));
      then txt;

    case ( txt,
           _ )
      then txt;
  end matchcontinue;
end mjmpopToStr;

end Emit;
```

C.4.5 PAMTRANS lexer.l

```
%{
#include <stdlib.h>
#define YYSTYPE void*
#include "parser.h"


#ifdef RML
#include "yacclib.h"
#include "Absyn.h"
#else
#include "meta_modelica.h"
extern struct record_description Absyn_Exp_INT__desc;
#define Absyn__INT(X1)       (mmc_mk_box2(3,&Absyn_Exp_INT__desc,X1))
#endif

int absyn_integer(char *s);
int absyn_ident_or_keyword(char *s);
int yywrap();
%}

%option yylineno

%x c_comment

whitespace   [ \t\n]+
letter       [a-zA-Z]
ident           {letter}({letter}|{digit})*
digit        [0-9]
digits       {digit}+
icon         {digits}
/* Lex style lexical syntax of tokens in the PAM language */

%%


{whitespace} ;
{ident}      return absyn_ident_or_keyword(yytext); /* T_IDENT */
{digits}     return absyn_integer(yytext);  /* T_INTCONST */
":="         return T_ASSIGN;
```

```
"+"          return T_ADD;
"-"          return T_SUB;
"*"          return T_MUL;
"/"          return T_DIV;
"("          return T_LPAREN;
")"          return T_RPAREN;
"<"          return T_LT;
"<="         return T_LE;
"="          return T_EQ;
"<>"         return T_NE;
">="         return T_GE;
">"          return T_GT;
";"          return T_SEMIC;


"/\*"       {
               BEGIN(c_comment);
             }
<c_comment>
{
    "\*/"    { BEGIN(INITIAL); }
    "/\*"    { yyerror("Suspicious comment"); }
    [^\n]    ;
    \n       ;
    <<EOF>>  {
         yyerror("Unterminated comment");
               yyterminate();
             }
}

%%

/* Make an RML integer from a C string representation (decimal),
   box it for our abstract syntax, put in yylval and return constant token. */

int absyn_integer(char *s)
{
  yylval = Absyn__INT(mmc_mk_icon(atoi(s)));
  return T_INTCONST;
}

/* Make an RML Ident or a keyword token from a C string */
/* Reserved words: if,then,else,endif,while,do,end,to,read,write */

static struct keyword_s
{
  char *name;
  int token;
} kw[] =
{
  {"do",        T_DO},
  {"else",      T_ELSE},
  {"end",       T_END},
  {"if",        T_IF},
  {"read",   T_READ},
  {"then",      T_THEN},
  {"while",     T_WHILE},
  {"write",     T_WRITE},
};

int absyn_ident_or_keyword(char *s)
{
        int low = 0;
        int high = (sizeof kw) / sizeof(struct keyword_s) - 1;

        while( low <= high ) {
                int mid = (low + high) / 2;
```

```
                int cmp = strcmp(kw[mid].name, yytext);
                if( cmp == 0 )
                {
                        return kw[mid].token;
                }
                else if( cmp < 0 )
                        low = mid + 1;
                else
                        high = mid - 1;
        }
        yylval = mmc_mk_scon(s);
        return T_IDENT;
}

int yywrap()
{
        return 1;
}
```

C.4.6 PAMTRANS parser.y

```
%{
#include <stdio.h>

void yyerror(char *str);
typedef void *rml_t;
#define YYSTYPE rml_t
rml_t absyntree;

void* parse()
{
  absyntree = NULL;
  yyparse();
  return absyntree;
}

#ifdef RML
#include "yacclib.h"
#include "Absyn.h"
#else
#include "meta_modelica.h"

/* Namedecl */
extern struct record_description Absyn_Decl_NAMEDECL__desc;

#define Absyn__NAMEDECL(X1,X2) (mmc_mk_box3(3,&Absyn_Decl_NAMEDECL__desc,X1,X2)
)

/* Program */
extern struct record_description Absyn_Prog_PROG__desc;

#define Absyn__PROG(X1,X2)     (mmc_mk_box3(3,&Absyn_Prog_PROG__desc,X1,X2))

/* BinOp */
extern struct record_description Absyn_BinOp_ADD__desc;
extern struct record_description Absyn_BinOp_SUB__desc;
extern struct record_description Absyn_BinOp_MUL__desc;
extern struct record_description Absyn_BinOp_DIV__desc;

#define Absyn__ADD (mmc_mk_box1(3,&Absyn_BinOp_ADD__desc))
#define Absyn__SUB (mmc_mk_box1(4,&Absyn_BinOp_SUB__desc))
#define Absyn__MUL (mmc_mk_box1(5,&Absyn_BinOp_MUL__desc))
#define Absyn__DIV (mmc_mk_box1(6,&Absyn_BinOp_DIV__desc))

/* RelOp */
extern struct record_description Absyn_RelOp_EQ__desc;
extern struct record_description Absyn_RelOp_GT__desc;
```

```
extern struct record_description Absyn_RelOp_LT__desc;
extern struct record_description Absyn_RelOp_LE__desc;
extern struct record_description Absyn_RelOp_GE__desc;
extern struct record_description Absyn_RelOp_NE__desc;

#define Absyn__EQ (mmc_mk_box1(3,&Absyn_RelOp_EQ__desc))
#define Absyn__GT (mmc_mk_box1(4,&Absyn_RelOp_GT__desc))
#define Absyn__LT (mmc_mk_box1(5,&Absyn_RelOp_LT__desc))
#define Absyn__LE (mmc_mk_box1(6,&Absyn_RelOp_LE__desc))
#define Absyn__GE (mmc_mk_box1(7,&Absyn_RelOp_GE__desc))
#define Absyn__NE (mmc_mk_box1(8,&Absyn_RelOp_NE__desc))

/* UnOp */
extern struct record_description Absyn_UnOp_NEG__desc;

#define Absyn__NEG (mmc_mk_box1(3,&Absyn_UnOp_NEG__desc))

/* Exp */
extern struct record_description Absyn_Exp_INT__desc;
extern struct record_description Absyn_Exp_IDENT__desc;
extern struct record_description Absyn_Exp_BINARY__desc;
extern struct record_description Absyn_Exp_RELATION__desc;

#define Absyn__INT(X1)            (mmc_mk_box2(3,&Absyn_Exp_INT__desc,X1))
#define Absyn__IDENT(X1)          (mmc_mk_box2(4,&Absyn_Exp_IDENT__desc,X1))
#define Absyn__BINARY(X1,OP,X2)   (mmc_mk_box4(5,&Absyn_Exp_BINARY__desc,X1,OP,
X2))
#define Absyn__RELATION(X1,OP,X2) (mmc_mk_box4(6,&Absyn_Exp_RELATION__desc,X1,O
P,X2))

/* Stmt */
extern struct record_description Absyn_Stmt_ASSIGN__desc;
extern struct record_description Absyn_Stmt_IF__desc;
extern struct record_description Absyn_Stmt_WHILE__desc;
extern struct record_description Absyn_Stmt_TODO__desc;
extern struct record_description Absyn_Stmt_READ__desc;
extern struct record_description Absyn_Stmt_WRITE__desc;
extern struct record_description Absyn_Stmt_SEQ__desc;
extern struct record_description Absyn_Stmt_SKIP__desc;

#define Absyn__ASSIGN(X1,X2)   (mmc_mk_box3(3,&Absyn_Stmt_ASSIGN__desc,X1,X2))
#define Absyn__IF(X1,X2,X3)    (mmc_mk_box4(4,&Absyn_Stmt_IF__desc,X1,X2,X3))
#define Absyn__WHILE(X1,X2)    (mmc_mk_box3(5,&Absyn_Stmt_WHILE__desc,X1,X2))
#define Absyn__TODO(X1,X2)     (mmc_mk_box3(6,&Absyn_Stmt_TODO__desc,X1,X2))
#define Absyn__READ(X1)        (mmc_mk_box2(7,&Absyn_Stmt_READ__desc,X1))
#define Absyn__WRITE(X1)       (mmc_mk_box2(8,&Absyn_Stmt_WRITE__desc,X1))
#define Absyn__SEQ(X1,X2)      (mmc_mk_box3(9,&Absyn_Stmt_SEQ__desc,X1,X2))
#define Absyn__SKIP            (mmc_mk_box1(10,&Absyn_Stmt_SKIP__desc))

#endif
%}

%token T_READ
%token T_WRITE
%token T_ASSIGN
%token T_IF
%token T_THEN
%token T_ENDIF
%token T_ELSE
%token T_TO
%token T_DO
%token T_END
%token T_WHILE
%token T_LPAREN
%token T_RPAREN
%token T_IDENT
%token T_INTCONST
```

```
%token T_EQ
%token T_LE
%token T_LT
%token T_GT
%token T_GE
%token T_NE
%token T_ADD
%token T_SUB
%token T_MUL
%token T_DIV
%token T_SEMIC

%%

/* Yacc BNF grammar of the PAM language */

program               :  series
                                { absyntree = $1; }
series                :  statement
                                { $$ = Absyn__SEQ($1, Absyn__SKIP); }
                      |  statement series
                                { $$ = Absyn__SEQ($1, $2); }

statement             :  input_statement T_SEMIC
                                { $$ = $1; }
                      |  output_statement T_SEMIC
                                { $$ = $1; }
                      |  assignment_statement T_SEMIC
                                { $$ = $1; }
                      |  conditional_statement
                                { $$ = $1; }
                      |  definite_loop
                                { $$ = $1; }
                      |  while_loop
                                { $$ = $1; }

input_statement       :  T_READ  variable_list
                                { $$ = Absyn__READ($2); }

output_statement      :  T_WRITE  variable_list
                                { $$ = Absyn__WRITE($2); }

variable_list         :  variable
                                { $$ = mmc_mk_cons($1, mmc_mk_nil()); }
                      |  variable variable_list
                                { $$ = mmc_mk_cons($1, $2); }

assignment_statement  :  variable  T_ASSIGN  expression
                                { $$ = Absyn__ASSIGN($1, $3); }

conditional_statement :  T_IF comparison T_THEN series T_ENDIF
                                { $$ = Absyn__IF($2, $4, Absyn__SKIP); }
                      |  T_IF comparison T_THEN series
                                        T_ELSE series T_ENDIF
                                { $$ = Absyn__IF($2, $4, $6); }

definite_loop         :  T_TO expression T_DO series T_END
                                { $$ = Absyn__TODO($2, $4); }

while_loop            :  T_WHILE comparison T_DO series T_END
                                { $$ = Absyn__WHILE($2, $4); }

expression       :  term
                                { $$ = $1; }
                 |  expression  weak_operator  term
                                { $$ = Absyn__BINARY($1, $2, $3); }
```

```
term             :  element
                                { $$ = $1; }
                 |  term  strong_operator  element
                                { $$ = Absyn__BINARY($1, $2, $3); }

element          :  constant
                                { $$ = $1; }
                 |  variable
                                { $$ = Absyn__IDENT($1); }
                 |  T_LPAREN  expression  T_RPAREN
                                { $$ = $2; }

comparison       :  expression  relation  expression
                                { $$ = Absyn__RELATION($1, $2, $3); }

variable         :  T_IDENT
                                { $$ = $1; }
constant         :  T_INTCONST
                                { $$ = $1; }

relation         : T_EQ { $$ = Absyn__EQ;}
                 | T_LE { $$ = Absyn__LE;}
                 | T_LT { $$ = Absyn__LT;}
                 | T_GT { $$ = Absyn__GT;}
                 | T_GE { $$ = Absyn__GE;}
                 | T_NE { $$ = Absyn__NE;}

weak_operator    : T_ADD { $$ = Absyn__ADD;}
                 | T_SUB { $$ = Absyn__SUB;}

strong_operator  : T_MUL { $$ = Absyn__MUL;}
                 | T_DIV { $$ = Absyn__DIV;}

%%

void yyerror(char *str) {
}
```

C.4.7 PAMTRANS Main.mo

```
package Main

public
import Mcode;
import Absyn;

protected
import Parse;
import Trans;
import Emit;

public
  function main
   "Parse and translate a PAM program into MCode,
    then emit it as textual assembly code."
    type Mcode_MCodeLst = list<Mcode.MCode>;
  protected
    Absyn.Stmt program;
    Mcode_MCodeLst mcode;
  algorithm
    print("[Parse. Enter a program, then press ");
    print("CTRL+z (Windows) or CTRL+d (Linux).]\n");
    program := Parse.parse();
    mcode := Trans.transProgram(program);
    print(Tpl.tplString(Emit.emitAssembly,mcode));
  end main;
```

```
end Main;
```

C.4.8 PAMTRANS Parse.mo

```
package Parse

import Absyn;

  function parse
    output Absyn.Stmt outStmt;
    external "C" outStmt = parse() annotation(Library={"lexer.o","parser.o"});
  end parse;

end Parse;
```

C.4.9 PAMTRANS Makefile

```
SOLUTIONS=SCRIPT.mos
CLEAN=Absyn_* Emit_* Main_* Mcode_* Parse_* Trans_*
DEPS=lexer.o parser.o Emit.mo Tpl.mo.dummy Print.mo.dummy Util.mo.dummy Debug.m
o.dummy RTOpts.mo.dummy System.mo.dummy Error.mo.dummy ErrorExt.mo.dummy
include ../common.omc

Tpl.mo.dummy:
        test -
f ../../../../Compiler/Template/Tpl.mo && cp ../../../../Compiler/Template/Tpl.
mo .

Print.mo.dummy:
        test -
f ../../../../Compiler/Util/Print.mo && cp ../../../../Compiler/Util/Print.mo .


Util.mo.dummy:
        test -
f ../../../../Compiler/Util/Util.mo && cp ../../../../Compiler/Util/Util.mo .

Debug.mo.dummy:
        test -
f ../../../../Compiler/Util/Debug.mo && cp ../../../../Compiler/Util/Debug.mo .


RTOpts.mo.dummy:
        test -
f ../../../../Compiler/Util/RTOpts.mo && cp ../../../../Compiler/Util/RTOpts.mo
 .

System.mo.dummy:
        test -
f ../../../../Compiler/Util/System.mo && cp ../../../../Compiler/Util/System.mo
 .

Error.mo.dummy:
        test -
f ../../../../Compiler/Util/Error.mo && cp ../../../../Compiler/Util/Error.mo .


ErrorExt.mo.dummy:
        test -
f ../../../../Compiler/Util/ErrorExt.mo && cp ../../../../Compiler/Util/ErrorEx
t.mo .
```

Appendix D

Exercises

D.1 Exercises – Introduction and Interpretive Semantics

D.1.1 Exercise 01_experiment – Types, Functions, Constants, Printing Values

In this exercise you will experiment with some MetaModelica language constructs:

- Types
- Constants
- Functions

Exercise: Write some functions in Functions.mo to display the complex constants defined in Types.mo. Search for `// your code here` in `Main.mo` and `Functions.mo`.

The solution is available in the file SOLUTION.txt and in Appendix E.1.

Hints:

- To build the project leave the input box empty when building the project.
- To run the application type "run" when building the project.

The files where you should add your code are `Main.mo` and `Functions.mo`, also shown below:

```
package Functions

import Types;

function test
  input String s;
  output Integer x;
algorithm
  x := matchcontinue s
    case "one"   then 1;
    case "two"   then 2;
    case "three" then 3;
    case _ then 0;
  end matchcontinue;
end test;

function factorial
  input Integer inValue;
  output Integer outValue;
algorithm
  outValue := matchcontinue inValue
    local Integer n;
    case 0 then 1;
    case n then n*factorial(n-1);
  end matchcontinue;
end factorial;

// your code here

end Functions;
```

```
package Main

import Types;
import Functions;

function main
 input list<String> arg;
algorithm
 _ := match arg
  local
    Integer i, n;
    String str, n_str;

  case (n_str::_)
   equation
     // factorial
     print("Factorial of " + n_str + " is: ");
     n = stringInt(n_str);
     i = Functions.factorial(n);
     str = intString(i);
     print(str);

     // test function
     print("\nCalling Functions.test(\"one\"):   " +
       intString(Functions.test("one")));
     print("\nCalling Functions.test(\"two\"):   " +
       intString(Functions.test("two")));
     print("\nCalling Functions.test(\"three\"): " +
       intString(Functions.test("three")));
     print("\nCalling Functions.test(\"other\"): " +
       intString(Functions.test("other")));

     // print Types.aliasConstant
     // your code here -- uncomment these when you wrote the functions
     // print("\nTypes.aliasConstant: ");
     // Functions.printAlias(Types.aliasConstant);

     // print Types.optionAliasConstant
     // your code here -- uncomment these when you wrote the functions
     // print("\nTypes.optionAliasConstant: ");
     // Functions.printOptionType(Types.optionAliasConstant);

     // print Types.optionAliasConstantNone
     // your code here -- uncomment these when you wrote the functions
     // print("\nTypes.optionAliasConstantNone: ");
     // Functions.printOptionType(Types.optionAliasConstantNone);

     // print Types.tupleConstant
     // your code here -- uncomment these when you wrote the functions
     // print("\nTypes.tupleConstant: ");
     // Functions.printTupleType(Types.tupleConstant);

     // print Types.listConstant
     // your code here -- uncomment these when you wrote the functions
     // print("\nTypes.listConstant: {");
     // Functions.printListType(Types.listConstant);
     // print("}");

     // print Types.oneRecord
     // your code here -- uncomment these when you wrote the functions
     // print("\nTypes.oneRecord: ");
     // Functions.printOneRecord(Types.oneRecord);

     // print Types.select
     // your code here -- uncomment these when you wrote the functions
     // print("\nTypes.select: ");
```

```
      // Functions.printSelect(Types.select);
    then ();
 end match;
end main;

end Main;
```

D.1.2 Exercise 02a_Exp1 – Adding New Features to a Small Language

In this exercise you will add new constructs to the Exp1 language by defining the evaluation semantics (interpretive semantics) of these constructs in MetaModelica..

Exercise: add the following constructs to the language

- A power operator
- A factorial operator
- Search for // your code here within exp1.mo

Note: the parser/lexer packages are ready, but give parser errors for the new operators until they are added in Exp1.mo (add them last in the uniontype to avoid weird errors).

A solution is available in the file SOLUTION.txt and in Appendix E.2.

Hints:

- To clean the project type "clean" when building the project.
- To build the project leave the input box empty when building the project.
- To run the calculator type "run" when building the project.
- If you need to edit the input of the calculator edit the file called `program.txt`.

The following is the package Exp1 where you should add your code, also available in the file Exp1.mo:

```
package Exp1

// Abstract syntax of the language Exp1
public
uniontype Exp "expressions"

  record INTconst
    Integer integer;
  end INTconst;

  record ADDop
    Exp exp1;
    Exp exp2;
  end ADDop;

  record SUBop
    Exp exp1;
    Exp exp2;
  end SUBop;

  record MULop
    Exp exp1;
    Exp exp2;
  end MULop;

  record DIVop
    Exp exp1;
    Exp exp2;
  end DIVop;

  record NEGop
```

```
    Exp exp;
  end NEGop;

  // Note that the external C code for the OMC version assumes you add the
  // records in a specific order.
  // The RML version doesn't make this assumtpion.

  // Add POWop here
  // your code here

  // Add FACop here
  // your code here

end Exp;

public function eval "Evaluation semantics  of Exp1"
  input Exp inExp;
  output Integer outInteger;
algorithm
  outInteger := matchcontinue (inExp)
    local
      Integer ival,v1,v2;
      Exp e1,e2,e;

    // evaluation of an integer node is the integer itself
    case (INTconst(integer = ival))  then ival;

    // Evaluation of an addition node PLUSop is v3, if v3 is the result of
    // adding the evaluated results of its children e1 and e2
    // Subtraction, multiplication, division operators have similar specs.
    case (ADDop(exp1 = e1,exp2 = e2))
      equation
        v1 = eval(e1);
        v2 = eval(e2);
      then v1 + v2;

    case (SUBop(exp1 = e1,exp2 = e2))
      equation
        v1 = eval(e1);
        v2 = eval(e2);
      then v1 - v2;

    case (MULop(exp1 = e1,exp2 = e2))
      equation
        v1 = eval(e1);
        v2 = eval(e2);
      then v1*v2;

    case (DIVop(exp1 = e1,exp2 = e2))
      equation
        v1 = eval(e1);
        v2 = eval(e2);
      then intDiv(v1,v2);

    case (NEGop(exp = e))
      equation
        v1 = eval(e);
      then -v1;

    // your code here
    // add evaluation handlers for the new operators

  end matchcontinue;
end eval;

// your code here
// add a factorial function
```

```
end Exp1;
```

D.1.3 Exercise 02b_Exp2 – Adding New Features to a Small Language

In this exercise you will explore a different way to model the Exp1 language using different Exp trees. Explore the Exp2.mo file and compare it with the Exp1.mo file.

Homework:

- Implement the assignments from 02a_Exp1 within 02b_Exp2. Note that you will have to add the new operators to the lexer and parser.

Hints:

- To clean the project type "clean" when building the project.
- To build the project leave the input box empty when building the project.
- To run the calculator type "run" when building the project.
- If you need to edit the input of the calculator edit the file called program.txt

A solution is available in Appendix E.

The package Exp2, available in Exp2.mo, should be modified and extended:

```
package Exp2

public
uniontype Exp
  record INT
    Integer integer;
  end INT;

  record BINARY
    Exp exp1;
    BinOp binOp2;
    Exp exp3;
  end BINARY;

  record UNARY
    UnOp unOp;
    Exp exp;
  end UNARY;
end Exp;

uniontype BinOp
  record ADD end ADD;
  record SUB end SUB;
  record MUL end MUL;
  record DIV end DIV;
  // Add POW here
end BinOp;

uniontype UnOp
  record NEG end NEG;
  // Add FAC here
end UnOp;

function eval
  input Exp inExp;
  output Integer outInteger;
algorithm
  outInteger := matchcontinue (inExp)
    local
      Integer ival,v1,v2,v3;
```

```
        Exp e1,e2,e;
        BinOp binop;
        UnOp unop;

      case (INT(integer = ival)) then ival;

      case (BINARY(exp1 = e1,binOp2 = binop,exp3 = e2))
        equation
          v1 = eval(e1);
          v2 = eval(e2);
          v3 = applyBinop(binop, v1, v2);
        then v3;

      case (UNARY(unOp = unop,exp = e))
        equation
          v1 = eval(e);
          v2 = applyUnop(unop, v1);
        then v2;
    end matchcontinue;
  end eval;

  protected
  function applyBinop
    input BinOp inBinOp1;
    input Integer inInteger2;
    input Integer inInteger3;
    output Integer outInteger;
  algorithm
    outInteger := matchcontinue (inBinOp1,inInteger2,inInteger3)
      local Integer v1,v2;
      case (ADD(),v1,v2) then v1+v2;
      case (SUB(),v1,v2) then v1-v2;
      case (MUL(),v1,v2) then v1*v2;
      case (DIV(),v1,v2) then intDiv(v1,v2);
    end matchcontinue;
  end applyBinop;

  function applyUnop
    input UnOp inUnOp;
    input Integer inInteger;
    output Integer outInteger;
  algorithm
    outInteger := match (inUnOp,inInteger)
      local Integer v;
      case (NEG(),v) then -v;
    end match;
  end applyUnop;

  end Exp2;
```

D.1.4 Exercise 03 Symbolic Derivative – Differentiating an Expression Using Symbolic Manipulation

In this exercise you transform expressions by symbolic differentiation.

Assignment:

- Add rules to derive ’-’, ’*’, sine, cosine and power expressions
- Add rules to simplify ’-’, sine, cosine and power expressions
- Search for `// your code here` within `SymbolicDerivative.mo`

The file `SOLUTION.txt` and Appendix E presents the solution to the assignment.

Hints:

- To clean the project type "clean" when building the project.

- To build the project leave the input box empty when building the project.
- To run the calculator type "run" when building the project.
- If you need to edit the input of the program edit the file called `program.txt`

```
package SymbolicDerivative

uniontype Exp

  record INT
    Integer integer;
  end INT;

  record ADD
    Exp exp1;
    Exp exp2;
  end ADD;

  record SUB
    Exp exp1;
    Exp exp2;
  end SUB;

  record MUL
    Exp exp1;
    Exp exp2;
  end MUL;

  record DIV
    Exp exp1;
    Exp exp2;
  end DIV;

  record NEG
    Exp exp;
  end NEG;

  record IDENT
    String id;
  end IDENT;

  record CALL
    String id;
    list<Exp> args;
  end CALL;

end Exp;

public function main
"Prints the expression and its derivative"
  input Exp expr;
  output Integer i;
protected
  Exp diffExpr;
  Exp simpleExpr;
algorithm
  print("f(x) = ");
  printExp(expr);
  print("\n");
  print("[Differentiating expression]\n");
  diffExpr := diff(expr,"x");
  print("f'(x) = ");
  printExp(diffExpr);
  print("\n");
  print("[Simplifying expression]\n");
  simpleExpr := simplifyExp(diffExpr);
```

```
    print("f'(x) = ");
    printExp(simpleExpr);
    print("\n");
    i := 0;
  end main;

  protected function diff
    input Exp expr;
    input String timevar;
    output Exp diffExpr;
  algorithm
    diffExpr := matchcontinue (expr,timevar)
      local
        String id,id1,id2;
        Exp e1prim,e2prim,e1,e2;

      // der of constant
      case (INT(_), _) then INT(0);

      // der of time variable
      case(IDENT(id1), id2)
        equation
          true = id1 ==& id2;
        then INT(1);

      // der of time-independent variable
      case (IDENT(_), _) then INT(0);

      // (e1+e2)' => e1'+e2'
      case (ADD(e1,e2),id)
        equation
          e1prim = diff(e1,id);
          e2prim = diff(e2,id);
        then ADD(e1prim,e2prim);

      // (e1/e2)' => (e1'*e2 - e1*e2')/e2*e2
      case (DIV(e1,e2),id)
        equation
          e1prim = diff(e1,id);
          e2prim = diff(e2,id);
        then DIV(SUB(MUL(e1prim,e2),MUL(e1,e2prim)), MUL(e2,e2));

      // (-e1)' => -e1'
      case (NEG(e1),id)
        equation
          e1prim = diff(e1,id);
        then NEG(e1prim);

      // your code here
      // (e1-e2)' => e1'-e2'
      // (e1*e2)' => e1'*e2 + e1*e2'
      // sin(e1)' => cos(e1)*e1'
      // cos(e1)' => -sin(e1)*e1'
      // pow(e1,INT(r))' => r*e1'*pow(e1,INT(r-1))

      // default case, e1' => e1'
      case (e1,_) then CALL("der",{e1});

    end matchcontinue;
  end diff;

  protected function simplifyExp
  "When differentating an expression, you often end up with lots of expressions
  that you can simplify (e.g. 1*x = x).
  simplifyExp simplifies leaf nodes first because if we did everything in one
  function we would get something like:
  (2*(0*sin(x))): (2*(0*sin(x))) => (2=>2 * (0*sin(x))=>0) => (2*0)
```

```
  But we want to do this:
  (2*(0*sin(x))): 2=>2, (0*sin(x)) => 0, (2*0) => 0"
    input Exp expr;
    output Exp simpleExpr;
  algorithm
    simpleExpr := matchcontinue (expr)
      local
        Exp e,e1,e2,sim1,sim2,res;
        String id;
        list<Exp> exprList,simpleExprList;

      case ADD(e1,e2)
        equation
          sim1 = simplifyExp(e1);
          sim2 = simplifyExp(e2);
          res = simplifyExp2(ADD(sim1,sim2));
        then res;

      case SUB(e1,e2)
        equation
          sim1 = simplifyExp(e1);
          sim2 = simplifyExp(e2);
          res = simplifyExp2(SUB(sim1,sim2));
        then res;

      case MUL(e1,e2)
        equation
          sim1 = simplifyExp(e1);
          sim2 = simplifyExp(e2);
          res = simplifyExp2(MUL(sim1,sim2));
        then res;

      case DIV(e1,e2)
        equation
          sim1 = simplifyExp(e1);
          sim2 = simplifyExp(e2);
          res = simplifyExp2(DIV(sim1,sim2));
        then res;

      case NEG(e1)
        equation
          sim1 = simplifyExp(e1);
          res = simplifyExp2(NEG(sim1));
        then res;

      case CALL(id,exprList)
        equation
          simpleExprList = simplifyExpList(exprList);
          res = simplifyExp2(CALL(id,simpleExprList));
        then res;

      case e then e; // IDENT, INT

    end matchcontinue;
  end simplifyExp;

  protected function simplifyExpList
    input list<Exp> exprList;
    output list<Exp> simpleExprList;
  algorithm
    simpleExprList := matchcontinue(exprList)
      local
        list<Exp> rest,simpleRest;
        Exp simpleExpr,expr;

      case {} then {};
```

```
      case expr::rest
        equation
          simpleExpr = simplifyExp(expr);
          simpleRest = simplifyExpList(rest);
        then simpleExpr :: simpleRest;

    end matchcontinue;
  end simplifyExpList;

  protected function simplifyExp2
    input Exp expr;
    output Exp simpleExpr;
  algorithm
    simpleExpr := matchcontinue (expr)
      local
        Exp e,e1,e2,sim1,sim2;
        Integer i,i1,i2;

      // Simplify addition
      case ADD(INT(i1),INT(i2)) equation i = i1 + i2; then INT(i);
      case ADD(INT(0),e) then e;
      case ADD(e,INT(0)) then e;

      // Simplify multiplication
      case MUL(INT(i1),INT(i2)) equation i = i1 * i2; then INT(i);
      case MUL(INT(0),e) then INT(0);
      case MUL(e,INT(0)) then INT(0);
      case MUL(INT(1),e) then e;
      case MUL(e,INT(1)) then e;

      // Simplify division
      // case DIV(INT(i1),INT(i2)) can give a real number, so don't simplify
      case DIV(e,INT(1)) then e;

      // Simplify some expressions of these types
      // SUB(INT,INT)
      // SUB(e,0)
      // SUB(0,e)
      // SUB(e1,NEG(e2))

      // NEG(INT)
      // NEG(NEG(e))

      // sin(0)
      // cos(0)
      // pow(x,0)
      // pow(x,1)

      // your code here

      // Default case, we can't simplify anymore
      case e then e;

    end matchcontinue;
  end simplifyExp2;

  // Functions for printing expressions

  public function printExp
    input Exp exp;
  protected
    String str;
  algorithm
    str := expStr(exp);
    print(str);
  end printExp;
```

```
protected function expStr
"Translates an Exp into a String"
  input Exp exp;
  output String str;
algorithm
  str := matchcontinue (exp)
    local
      Integer i;
      Exp e,lhs,rhs;
      String left,right,res,id;
      list<Exp> expList;

    case INT(i) then intString(i);
    case ADD(lhs,rhs) then binExpStr(lhs,"+",rhs);
    case SUB(lhs,rhs) then binExpStr(lhs,"-",rhs);
    case MUL(lhs,rhs) then binExpStr(lhs,"*",rhs);
    case DIV(lhs,rhs) then binExpStr(lhs,"/",rhs);
    case NEG(e) then "(-" + expStr(e) + ")";
    case IDENT(id) then id;

    case CALL("der",{e})
      equation
        res = expStr(e);
      then "(" + res + ")'";

    case CALL("pow",{e,INT(i)})
      equation
        res = expStr(e);
      then res + "^" + intString(i);

    case CALL(id,expList)
      equation
        res = expListStr(expList);
      then id + "(" + res + ")";

    case _ then "#UNKNOWN_EXP#";

  end matchcontinue;
end expStr;

protected function expListStr
"Translates a list of Exp into a comma-separated String"
  input list<Exp> expList;
  output String str;
algorithm
  str := matchcontinue (expList)
    local
      Exp e;
      list<Exp> rest;
      String res_1,res_2;

    case {} then "";

    case {e} then expStr(e);

    case e::rest
      equation
        res_1 = expStr(e);
        res_2 = expListStr(rest);
      then res_1 + "," + res_2;

  end matchcontinue;
end expListStr;

protected function binExpStr
"Translates a binary expression (lhs op rhs) into a String"
  input Exp lhs;
```

```
    input String op;
    input Exp rhs;
    output String str;
  algorithm
    str := "(" + expStr(lhs) + op + expStr(rhs) + ")";
  end binExpStr;

  end SymbolicDerivative;
```

D.1.5 Exercise 04 Assignment – Printing AST and Environments

04_assignment exercise

ASSIGNMENT:

In this exercise you will add a new functions to print:

- - the assignments present in the current program before the actual evaluation
- - the environment after it was augmented with the assignments
- - search for // your code here within Assignment.mo

SOLUTION.txt and Appendix E presents the solution to the assignment.

Hints:

- To clean the project type "clean" when building the project.
- To build the project leave the input box empty when building the project.
- To run the calculator type "run" when building the project.
- If you need to edit the input of the calculator edit the file called program.txt

```
package Assignment "Assignment.mo"

public
type ExpLst = list<Exp> "list of expressions";

// Abstract syntax for the Assignments language
uniontype Program "a program"

  record PROGRAM
    ExpLst expLst;
    Exp exp;
  end PROGRAM;

end Program;

uniontype Exp "expressions"

  record INT
    Integer integer;
  end INT;

  record BINARY
    Exp exp1;
    BinOp binOp2;
    Exp exp3;
  end BINARY;

  record UNARY
    UnOp unOp;
    Exp exp;
  end UNARY;

  record ASSIGN
    Ident ident;
```

```
    Exp exp;
  end ASSIGN;

  record IDENT
    Ident ident;
  end IDENT;

end Exp;

public
uniontype BinOp
  record ADD end ADD;
  record SUB end SUB;
  record MUL end MUL;
  record DIV end DIV;
end BinOp;

public
uniontype UnOp
  record NEG end NEG;

end UnOp;

public
type Ident = String "an identifier is just a string";

type Value = Integer "Values stored in environments";

type VarBnd = tuple<Ident,Value> "a binding is a tuple of id and value";

type Env = list<VarBnd> "environment is a list of bindings";

protected function lookup
 "lookup returns the value associated with an identifier.
  If no association is present, lookup will fail."
  input Env inEnv;
  input Ident inIdent;
  output Value outValue;
algorithm
  outValue := matchcontinue (inEnv,inIdent)
    local
      Ident id2,id;
      Value value;
      Env rest;
    case ((id2,value) :: rest,id)
      then if id == id2
           then value
           else lookup(rest, id);
  end matchcontinue;
end lookup;

protected function lookupextend
  input Env inEnv;
  input Ident inIdent;
  output Env outEnv;
  output Value outValue;
algorithm
  (outEnv,outValue) := matchcontinue (inEnv,inIdent)
    local
      Env env;
      Ident id;
      Value value;

    case (env,id)
      equation
        failure(_ = lookup(env, id));
```

```
      then ((id,0) :: env,0);

    case (env,id)
      equation
        value = lookup(env, id);
      then (env,value);

  end matchcontinue;
end lookupextend;

protected function update
  input Env env;
  input Ident id;
  input Value value;
  output Env outEnv;
algorithm
  outEnv := (id,value) :: env;
end update;

protected function applyBinop
  input BinOp inBinOp1;
  input Integer inInteger2;
  input Integer inInteger3;
  output Integer outInteger;
algorithm
  outInteger := matchcontinue (inBinOp1,inInteger2,inInteger3)
    local Value v1,v2;
    case (ADD(),v1,v2) then v1+v2;
    case (SUB(),v1,v2) then v1-v2;
    case (MUL(),v1,v2) then v1*v2;
    case (DIV(),v1,v2) then intDiv(v1,v2);
  end matchcontinue;
end applyBinop;

protected function applyUnop
  input UnOp inUnOp;
  input Integer inInteger;
  output Integer outInteger;
algorithm
  outInteger := match (inUnOp,inInteger)
    local Value v;
    case (NEG(),v) then -v;
  end match;
end applyUnop;

protected function eval
  input Env inEnv;
  input Exp inExp;
  output Env outEnv;
  output Integer outInteger;
algorithm
  (outEnv,outInteger) := matchcontinue (inEnv,inExp)
    local
      Env env,env2,env3,env1;
      Value ival,value,v1,v2,v3;
      Ident s,id;
      Exp exp,e1,e2,e;
      BinOp binop;
      UnOp unop;

    case (env,INT(integer = ival)) then (env,ival);

    // eval of an identifier node will lookup the identifier and return a
    // value if present; otherwise insert a binding to zero, and return zero.
    case (env,IDENT(ident = id))
      equation
        (env2,value) = lookupextend(env, id);
```

```
      then (env2,value);

    // eval of an assignment node returns the updated
    // environment and the assigned value.
    case (env,ASSIGN(ident = id,exp = exp))
      equation
        (env2,value) = eval(env, exp);
        env3 = update(env2, id, value);
      then (env3,value);

    // eval of a node e1,ADD,e2 , etc. in an environment env
    case (env1,BINARY(exp1 = e1,binOp2 = binop,exp3 = e2))
      equation
        (env2,v1) = eval(env1, e1);
        (env3,v2) = eval(env2, e2);
        v3 = applyBinop(binop, v1, v2);
      then (env3,v3);

    case (env1,UNARY(unOp = unop,exp = e))
      equation
        (env2,v1) = eval(env1, e);
        v2 = applyUnop(unop, v1);
      then (env2,v2);

  end matchcontinue;
end eval;

protected function evals
  input Env inEnv;
  input ExpLst inExpLst;
  output Env outEnv;
algorithm
  outEnv := matchcontinue (inEnv,inExpLst)
    local
      Env e,env2,env3,env;
      Value v;
      Ident s;
      Exp exp;
      ExpLst expl;

    // the environment stays the same if there are no expressions
    case (e,{}) then e;

    // the head expression is evaluated in the current environment
    // generating a new environment in which the rest of the expression
    // list is evaluated. the last environment is returned
    case (env,exp :: expl)
      equation
        (env2,v) = eval(env, exp);
        env3 = evals(env2, expl);
      then env3;

  end matchcontinue;
end evals;

public function evalprogram
  input Program inProgram;
  output Integer outInteger;
algorithm
  outInteger := match (inProgram)
    local
      ExpLst assignments_1,assignments;
      Env env2;
      Value value;
      Exp exp;
```

```
    case (PROGRAM(expLst = assignments,exp = exp))
      equation
        assignments_1 = listReverse(assignments);
        // your code here -> print assignments_1 and exp
        // print("The assignments: ");
        // printAssignments(assignments_1);
        // print("The expression: ");
        // printAssignments({exp});
        env2 = evals({}, assignments_1);
        // your code here -> print env2
        // print("The environment: ");
        // printEnvironment(env2);
        (_,value) = eval(env2, exp);
      then value;

  end match;
end evalprogram;

// your code here

end Assignment;
```

D.1.6 Exercise 04a_AssignTwoType – Adding a New Type to a Language

In this exercise you will:

- Add a new String type which can hold only integers as strings to the current Exp node
- Add cases to evaluate expressions and assignments of type "2" + 1 + "1" + 1.0 in the eval function
- Search for `// your code here` within `AssignTwoType.mo`.

Optional exercise:

- Change the code to allow the use of identifiers before actual declaration

Example: a program in the AssignTwoType language of the form:

```
a := b + 1
b := 3
;
a+b
```

should return 7 instead of 4 as it returns now .

NOTE: the parser/lexer are ready, but you have to implement the types before they start parsing STRING properly.

The solution is available in the file SOLUTION.txt and in Appendix E.

Hints:

- To clean the project type "clean" when building the project.
- To build the project leave the input box empty when building the project.
- To run the calculator type "run" when building the project.
- If you need to edit the input of the calculator edit the file called program.txt

You should insert your code as marked below:

```
package AssignTwoType "file AssignTwoType.mo"

public
type ExpLst = list<Exp> "a list of expresions";
```

```
// Abstract syntax for the Assigntwotype language
uniontype Program ”a program in our language”
  record PROGRAM
    ExpLst expLst;
    Exp exp;
  end PROGRAM;
end Program;

uniontype Exp "expressions in our language"

  record INT
    Integer integer;
  end INT;

  record REAL
    Real real;
  end REAL;

  record BINARY
    Exp exp1;
    BinOp binOp2;
    Exp exp3;
  end BINARY;

  record UNARY
    UnOp unOp;
    Exp exp;
  end UNARY;

  record ASSIGN
    Ident ident;
    Exp exp;
  end ASSIGN;

  record IDENT
    Ident ident;
  end IDENT;

  // your code here
  // add a record called STRING

end Exp;

uniontype BinOp
  record ADD end ADD;
  record SUB end SUB;
  record MUL end MUL;
  record DIV end DIV;
end BinOp;

uniontype UnOp
  record NEG end NEG;
end UnOp;

type Ident = String;

uniontype Value "Values stored in environments"
  record INTval
    Integer integer;
  end INTval;

  record REALval
    Real real;
  end REALval;
end Value;
```

```
type VarBnd = tuple<Ident,Value> "Bindings and environments";

type Env = list<VarBnd>;

uniontype Ty2
  "Ty2 is an auxiliary datatype used to handle types during evaluation"

  record INT2
    Integer integer1;
    Integer integer2;
  end INT2;

  record REAL2
    Real real1;
    Real real2;
  end REAL2;

end Ty2;

protected function printvalue
  input Value inValue;
algorithm
  _ := matchcontinue (inValue)
    local
      Ident str;
      Integer i;
      Real r;

    case (INTval(integer = i))
      equation
        str = intString(i);
        print(str);
      then ();

    case (REALval(real = r))
      equation
        str = realString(r);
        print(str);
      then ();

  end matchcontinue;
end printvalue;

public function evalprogram
  input Program inProgram;
algorithm
  _ := matchcontinue (inProgram)
    local
      ExpLst assignments_1,assignments;
      Env env2;
      Value value;
      Exp exp;

    case (PROGRAM(expLst = assignments,exp = exp))
      equation
        assignments_1 = listReverse(assignments);
        env2 = evals({}, assignments_1);
        (_,value) = eval(env2, exp);
        printvalue(value);
        print("\n");
      then ();

  end matchcontinue;
end evalprogram;

protected function evals
  input Env inEnv;
```

```
  input ExpLst inExpLst;
  output Env outEnv;
algorithm
  outEnv := matchcontinue (inEnv,inExpLst)
    local
      Env e,env2,env3,env;
      Exp exp;
      ExpLst expl;

    case (e,{}) then e;

    case (env,exp :: expl)
      equation
        (env2,_) = eval(env, exp);
        env3 = evals(env2, expl);
      then env3;

  end matchcontinue;
end evals;

protected function eval
  input Env inEnv;
  input Exp inExp;
  output Env outEnv;
  output Value outValue;
algorithm
  (outEnv,outValue) := matchcontinue (inEnv,inExp)
    local
      Env env,env2,env1;
      Integer ival,x,y,z;
      Real rval,rx,ry,rz;
      String sval;
      Value value,v1,v2;
      Ident id;
      Exp e1,e2,e,exp;
      BinOp binop;
      UnOp unop;

    // handle integers
    case (env,INT(integer = ival)) then (env,INTval(ival));

    // handle reals
    case (env,REAL(real = rval)) then (env,REALval(rval));

    // your code here
    // case (env, STRING(...)) ...

    // variable id
    case (env,IDENT(ident = id))
      equation
        (env2,value) = lookupextend(env, id);
      then (env2,value);

    // int binop int
    case (env,BINARY(exp1 = e1,binOp2 = binop,exp3 = e2))
      equation
        (env1,v1) = eval(env, e1);
        (env2,v2) = eval(env, e2);
        INT2(integer1 = x,integer2 = y) = typeLub(v1, v2);
        z = applyIntBinop(binop, x, y);
      then (env2,INTval(z));

    // int/real binop int/real
    case (env,BINARY(exp1 = e1,binOp2 = binop,exp3 = e2))
      equation
        (env1,v1) = eval(env, e1);
        (env2,v2) = eval(env, e2);
```

```
        REAL2(real1 = rx,real2 = ry) = typeLub(v1, v2);
        rz = applyRealBinop(binop, rx, ry);
      then (env2,REALval(rz));

    // int unop exp
    case (env,UNARY(unOp = unop,exp = e))
      equation
        (env1,INTval(integer = x)) = eval(env, e);
        y = applyIntUnop(unop, x);
      then (env1,INTval(y));

    // real unop exp
    case (env,UNARY(unOp = unop,exp = e))
      equation
        (env1,REALval(real = rx)) = eval(env, e);
        ry = applyRealUnop(unop, rx);
      then (env1,REALval(ry));

    // eval of an assignment node returns the updated
    // environment and the assigned value id := exp
    case (env,ASSIGN(ident = id,exp = exp))
      equation
        (env1,value) = eval(env, exp);
        env2 = update(env1, id, value);
      then (env2,value);

  end matchcontinue;
end eval;

protected function typeLub
  input Value inValue1;
  input Value inValue2;
  output Ty2 outTy2;
algorithm
  outTy2 := matchcontinue (inValue1,inValue2)
    local
      Integer x,y;
      Real x2,y2;

    case (INTval(integer = x),INTval(integer = y)) then INT2(x,y);

    case (INTval(integer = x),REALval(real = y2))
      equation
        x2 = intReal(x);
      then REAL2(x2,y2);

    case (REALval(real = x2),INTval(integer = y))
      equation
        y2 = intReal(y);
      then REAL2(x2,y2);

    case (REALval(real = x2),REALval(real = y2))
      then REAL2(x2,y2);

  end matchcontinue;
end typeLub;

protected function applyIntBinop
  input BinOp inBinOp1;
  input Integer inInteger2;
  input Integer inInteger3;
  output Integer outInteger;
algorithm
  outInteger := matchcontinue (inBinOp1,inInteger2,inInteger3)
    local Integer x,y;
    case (ADD(),x,y) then x + y;
    case (SUB(),x,y) then x - y;
```

```
    case (MUL(),x,y) then x * y;
    case (DIV(),x,y) then intDiv(x, y);
  end matchcontinue;
end applyIntBinop;

protected function applyRealBinop
  input BinOp inBinOp1;
  input Real inReal2;
  input Real inReal3;
  output Real outReal;
algorithm
  outReal := matchcontinue (inBinOp1,inReal2,inReal3)
    local Real x,y;
    case (ADD(),x,y) then x + y;
    case (SUB(),x,y) then x - y;
    case (MUL(),x,y) then x * y;
    case (DIV(),x,y) then x / y;
  end matchcontinue;
end applyRealBinop;

protected function applyIntUnop
  input UnOp inUnOp;
  input Integer inInteger;
  output Integer outInteger;
algorithm
  outInteger := matchcontinue (inUnOp,inInteger)
    local Integer x;
    case (NEG(),x) then -x;
  end matchcontinue;
end applyIntUnop;

protected function applyRealUnop
  input UnOp inUnOp;
  input Real inReal;
  output Real outReal;
algorithm
  outReal := matchcontinue (inUnOp,inReal)
    local Real x;
    case (NEG(),x) then realNeg(x);
  end matchcontinue;
end applyRealUnop;

protected function lookup
  input Env inEnv;
  input Ident inIdent;
  output Value outValue;
algorithm
  outValue := matchcontinue (inEnv,inIdent)
    local
      Ident id2,id;
      Value value;
      Env rest;

    // lookup returns the value associated with an identifier.
    // If no association is present, lookup will fail.
    // Identifier id is found in the first pair of the list,
    // and value is returned."
    case ((id2,value) :: _,id)
      equation
        equality(id = id2);
      then value;

    // id is not found in the first pair of the list, and lookup will
    // recursively search the rest of the list. If found, value is returned.
    case ((id2,_) :: rest,id)
      equation
```

```
        failure(equality(id = id2));
        value = lookup(rest, id);
      then value;
  end matchcontinue;
end lookup;

protected function lookupextend
  input Env inEnv;
  input Ident inIdent;
  output Env outEnv;
  output Value outValue;
algorithm
  (outEnv,outValue) := matchcontinue (inEnv,inIdent)
    local
      Value value;
      Env env;
      Ident id;

    // Return value of id in env. If id not present, add id and return 0
    case (env,id)
      equation
        failure(value = lookup(env, id));
        value = INTval(0);
      then ((id,value) :: env,value);

    case (env,id)
      equation
        value = lookup(env, id);
      then (env,value);

  end matchcontinue;
end lookupextend;

protected function update
  input Env inEnv;
  input Ident inIdent;
  input Value inValue;
  output Env outEnv;
algorithm
  outEnv := matchcontinue (inEnv,inIdent,inValue)
    local
      Env env;
      Ident id;
      Value value;
    case (env,id,value) then (id,value) :: env;
  end matchcontinue;
end update;

end AssignTwoType;
```

D.1.7 Exercise 04b_ModAssigntwotype – Modularized Specification

In this exercise you will explore a different way to structure your code within different packages. The code from 04a_assigntwotype is now split over 4 packages. Otherwise the exercise is the same. See Appendix E for a solution.

D.2 Exercises – Translational Semantics

D.2.1 Exercise 09_pamtrans – Small Translational Semantics

Additional example exercise that translates the Pam language with declarations to machine code, i.e. a translational semantics specification of a compiler rather than an interpreter as in the previous exercises. See Appendix E.10 for a solution.

Hints:

- To clean the project type "clean" when building the project.
- To build the project leave the input box empty when building the project.
- To run the calculator type "run" when building the project.
- If you need to edit the input of the pamTrans translator edit the file called `program.txt`.

D.2.2 Exercise 10_Petrol – Large Translational Semantics

Additional example exercise showing a translational semantics for a Pascal-like language called Petrol, essentially a subset of Pascal extended with pointer arithmetics. This gives a compiler from Petrol to C. See Appendix E.11 for a solution.

Hints:

- To clean the project type "clean" when building the project.
- To build the project leave the input box empty when building the project.
- To run the calculator type "run" when building the project.
- If you need to edit the input of the Petrol translator edit the Makefile.run target. Right now the Makefile.run target calls the petrol compiler with testd/big.d as input. There are additional example programs in testd and testp directories.

D.3 Exercises – Advanced

D.3.1 Exercise 05_advanced – Polymorphic Types and Higher Order Functions

In this exercise you will experiment with MetaModelica:

- Polymorphic types
- Constants
- Higher order functions

Exercises:

1. Write a polymorphic function that orders a list of any type (`Integer`, `String`, `Real` is enough). The function has as input a list and a compare function between the objects of that list. Also write the comparison functions for Integers, Strings and Reals. Test your function on the `Types.intList`

2. Write a polymorphic map function that applies a function over a list and returns a new list with the result. Write three functions that transform from:
- `Integer` to `Real`
- `Integer` to `String`
- `Real` to `String`

Use your map function and the two transformation functions to transform the `Types.intList` to a list of reals and a list of string, then apply the ordering function from point 1.

3. Write a polymorphic map function that applies a print function over a list (of Strings) and prints the it. Use the transformer functions from `Real->String` and `Integer->String` from point 2 to transform the `Real` list or the `Integer` list to a `String` list for printing.

A solution is available in the file `SOLUTION.txt` and in Appendix E.

Hints:

- To clean the project type "clean" when building the project.
- To build the project leave the input box empty when building the project.
- To run the application type "run" when building the project.

Appendix E

Solutions to Exercises

E.1 Solution 01_experiment – Types, Functions, Constants, Printing Values

The solution: updated functions for printing have been added to `Functions.mo`.

```
package Functions

import Types;

function test
  input String s;
  output Integer x;
algorithm
  x := matchcontinue s
    case "one"   then 1;
    case "two"   then 2;
    case "three" then 3;
    case _ then 0;
  end matchcontinue;
end test;

function factorial
  input Integer inValue;
  output Integer outValue;
algorithm
  outValue := matchcontinue inValue
    local Integer n;
    case 0 then 1;
    case n then n*factorial(n-1);
  end matchcontinue;
end factorial;

// an alias for the Real type
// type Alias = Real;
// constant Alias aliasConstant = 1.0;

function printAlias
  input Types.Alias aliasVariable;
algorithm
  print(realString(aliasVariable));
end printAlias;

// an option type which can be SOME(Alias) or NONE()
// type OptionType = Option<Alias>;
// constant OptionType optionAliasConstant = SOME(aliasConstant);

function printOptionType
  input Types.OptionType oVar;
algorithm
  _ := matchcontinue(oVar)
    local Types.Alias alias;

    case NONE()
      equation
```

```
        print("NONE()");
      then ();

    case SOME(alias)
      equation
        printAlias(alias);
      then ();

  end matchcontinue;
end printOptionType;

// a tuple type with 3 elements
//type TupleType = tuple<String, Alias, OptionType>;
//constant TupleType tupleConstant =
//  ("a tuple element", aliasConstant, optionAliasConstant);

function printTupleType
  input Types.TupleType tupleVar;
algorithm
  _ := match (tupleVar)
    local
      Types.Alias alias;
      Types.OptionType optionAlias;
      String str;

    case ((str, alias, optionAlias))
      equation
        print("(");
        print("\"" + str + "\"");
        print(", ");
        printAlias(alias);
        print(", ");
        printOptionType(optionAlias);
        print(")");
      then ();

  end match;
end printTupleType;


// a list type
//type ListType = list<TupleType>;
//constant ListType listConstant =
// {tupleConstant, ("another element", 2.0, NONE())};

function printListType
  input Types.ListType listVar;
algorithm
  _ := matchcontinue(listVar)
    local
      Types.TupleType element;
      Types.ListType rest;
      String str;

    case ({}) then ();

    case (element::{})
      equation
        printTupleType(element);
      then ();

    case (element::rest)
      equation
        printTupleType(element);
        print(", ");
        printListType(rest);
```

```
      then ();

  end matchcontinue;
end printListType;


// complex record types
//record OneRecord
//  String k;
//  Alias z;
//end OneRecord;
//constant OneRecord oneRecord = OneRecord("first element", 3.0);

function printOneRecord
  input Types.OneRecord oneRecordVar;
algorithm
  _ := match (oneRecordVar)
    local
      String cmp1;
      Types.Alias cmp2;

    case (Types.OneRecord(cmp1, cmp2))
      equation
        print("OneRecord(");
        print("\"" + cmp1 + "\"");
        print(", ");
        printAlias(cmp2);
        print(")");
      then ();

  end match;
end printOneRecord;


// complex uniontypes
//uniontype Select

//  record FirstAlternative
//    String x1;
//    String x2;
//  end FirstAlternative;

//  record SecondAlternative
//    Select x1;
//    Select x2;
//  end SecondAlternative;
//
//  record ThirdAlternative
//    Select next;
//  end ThirdAlternative;
//end Select;

//constant Select select =
//    ThirdAlternative(
//      SecondAlternative(
//        FirstAlternative("one", "First"),
//        FirstAlternative("two", "Second")));

function printSelect
  input Types.Select selectVar;
algorithm
  _ := matchcontinue(selectVar)
    local
      String cmp1, cmp2;
      Types.Select sel1, sel2;
```

```
    case (Types.FirstAlternative(cmp1, cmp2))
      equation
        print("FirstAlternative(");
        print("\"" + cmp1 + "\"");
        print(", ");
        print("\"" + cmp2 + "\"");
        print(")");
      then ();

    case (Types.SecondAlternative(sel1, sel2))
      equation
        print("SecondAlternative(");
        printSelect(sel1);
        print(", ");
        printSelect(sel2);
        print(")");
      then ();

    case (Types.ThirdAlternative(sel1))
      equation
        print("ThirdAlternative(");
        printSelect(sel1);
        print(")");
      then ();

  end matchcontinue;
end printSelect;

end Functions;

package Main

import Types;
import Functions;

function main
 input list<String> arg;
algorithm
 _ := match arg
  local
    Integer i, n;
    String str, n_str;

  case (n_str::_)
   equation
    // factorial
    print("Factorial of " + n_str + " is: ");
    n = stringInt(n_str);
    i = Functions.factorial(n);
    str = intString(i);
    print(str);

    // test function
    print("\nCalling Functions.test(\"one\"):   " +
      intString(Functions.test("one")));
    print("\nCalling Functions.test(\"two\"):   " +
      intString(Functions.test("two")));
    print("\nCalling Functions.test(\"three\"): " +
      intString(Functions.test("three")));
    print("\nCalling Functions.test(\"other\"): " +
      intString(Functions.test("other")));

    // print Types.aliasConstant
    print("\nTypes.aliasConstant: ");
    Functions.printAlias(Types.aliasConstant);

    // print Types.optionAliasConstant
```

```
      print("\nTypes.optionAliasConstant: ");
      Functions.printOptionType(Types.optionAliasConstant);

      // print Types.optionAliasConstantNone
      print("\nTypes.optionAliasConstantNone: ");
      Functions.printOptionType(Types.optionAliasConstantNone);

      // print Types.tupleConstant
      print("\nTypes.tupleConstant: ");
      Functions.printTupleType(Types.tupleConstant);

      // print Types.listConstant
      print("\nTypes.listConstant: {");
      Functions.printListType(Types.listConstant);
      print("}");

      // print Types.oneRecord
      print("\nTypes.oneRecord: ");
      Functions.printOneRecord(Types.oneRecord);

      // print Types.select
      print("\nTypes.select: ");
      Functions.printSelect(Types.select);
    then ();
  end match;
 end main;

 end Main;
```

E.2 Solution 02a_Exp1 – Adding New Features to a Small Language

The following changes are needed:

- parser.y file changes:

Locate the `uncomment here` section and remove the comment to make the comment active.

- Exp1.mo file changes:

```
Exp1.Exp type addition:
-----------------------
  record FACop
    Exp exp;
  end FACop;

  record POWop
    Exp exp1;
    Exp exp2;
  end POWop;


Exp1.eval function addition:
----------------------------

    case (FACop(exp = e))
      equation
        v1 = eval(e);
        v2 = fac(v1);
      then v2;

    case (POWop(exp1 = e1, exp2 = e2))
      local
        Integer v3;
      equation
        v1 = eval(e1);
        v2 = eval(e2);
```

```
        v3 = realInt(intReal(v1) ^. intReal(v2));
      then v3;

Exp1.fac new function:
----------------------
function fac
  input Integer i;
  output Integer o;
algorithm
  o := matchcontinue (i)

    case (0) then 1;

    case (n)
      local Integer n;
      then n*fac(n-1);

  end matchcontinue;
end fac;
```

E.3 Solution 03 Symbolic Derivative – Differentating an Expression Using Symbolic Manipulation

```
package SymbolicDerivative

uniontype Exp " expressions"

  record INT "literal integers"
    Integer integer;
  end INT;

  record ADD "additions"
    Exp exp1;
    Exp exp2;
  end ADD;

  record SUB "subtractions"
  Exp exp1;
    Exp exp2;
  end SUB;

  record MUL "multiplications"
    Exp exp1;
    Exp exp2;
  end MUL;

  record DIV "divisions"
    Exp exp1;
    Exp exp2;
  end DIV;

  record NEG "negation"
    Exp exp;
  end NEG;

  record IDENT "identifiers"
    String id;
  end IDENT;
```

```
    record CALL "function calls"
      String id;
      list<Exp> args;
    end CALL;
  end Exp;

  public function main "Prints the expression and its derivative"
    input Exp expr;
    output Integer i;
  protected
    Exp diffExpr;
    Exp simpleExpr;
  algorithm
    print("f(x) = "); printExp(expr); print("\n");
    print("[Differentiating expression]\n");
    diffExpr := diff(expr, "x");
    print("f'(x) = "); printExp(diffExpr); print("\n");
    print("[Simplifying expression]\n");
    simpleExpr := simplifyExp(diffExpr);
    print("f'(x) = "); printExp(simpleExpr); print("\n");
    i := 0;
  end main;

  protected function diff
    input Exp expr;
    input String timevar;
    output Exp diffExpr;
  algorithm
    diffExpr := matchcontinue (expr,timevar)
      local
        String id,id1,id2;
        Exp e1prim,e2prim,e1,e2;
        Integer i1,i2;

      // der of constant
      case (INT(_), _) then INT(0);

      // der of time variable
      case(IDENT(id1), id2)
        equation
          true = id1 ==& id2;
        then INT(1);

      // der of time-independent variable
      case (IDENT(_), _) then INT(0);

      // (e1+e2)' => e1'+e2'
      case (ADD(e1,e2),id)
        equation
          e1prim = diff(e1,id);
          e2prim = diff(e2,id);
        then ADD(e1prim,e2prim);

      // (e1/e2)' => (e1'*e2 - e1*e2')/e2*e2
      case (DIV(e1,e2),id)
        equation
          e1prim = diff(e1,id);
          e2prim = diff(e2,id);
        then DIV(SUB(MUL(e1prim,e2),MUL(e1,e2prim)), MUL(e2,e2));

      // (-e1)' => -(e1')
      case (NEG(e1),id)
        equation
          e1prim = diff(e1,id);
        then NEG(e1prim);

      // your code here
```

```
    // (e1-e2)' => e1'+e2'
    case (SUB(e1,e2),id)
      equation
        e1prim = diff(e1,id);
        e2prim = diff(e2,id);
      then SUB(e1prim,e2prim);

    // (e1*e2)' => e1'*e2 + e1*e2'
    case (MUL(e1,e2),id)
      equation
        e1prim = diff(e1,id);
        e2prim = diff(e2,id);
      then ADD(MUL(e1prim,e2),MUL(e1,e2prim));

    // sin(e1)' => cos(e1)*e1'
    case (CALL("sin", {e1}),id)
      equation
        e1prim = diff(e1,id);
      then MUL(CALL("cos",{e1}),e1prim);

    // cos(e1)' => -sin(e1)*e1'
    case (CALL("cos", {e1}),id)
      equation
        e1prim = diff(e1,id);
      then NEG(MUL(CALL("sin",{e1}),e1prim));

    // pow(e1,INT(i))' => i*e1'*pow(e1,INT(i-1))
    case (CALL("pow", {e1,INT(i1)}),id)
      equation
        e1prim = diff(e1,id);
        i2 = i1-1;
      then MUL(INT(i1),MUL(e1prim,CALL("pow",{e1,INT(i2)})));

    // default case, e1' => e1'
    case (e1,_) then CALL("der",{e1});

  end matchcontinue;
end diff;

protected function simplifyExp
"When differentating an expression, you often end up with lots of expressions
that you can simplify (e.g. 1*x = x).
simplifyExp simplifies leaf nodes first because if we did everything in one
function we would get something like:
(2*(0*sin(x))): (2*(0*sin(x))) => (2=>2 * (0*sin(x))=>0) => (2*0)
But we want to do this:
(2*(0*sin(x))): 2=>2, (0*sin(x)) => 0, (2*0) => 0"
  input Exp expr;
  output Exp simpleExpr;
algorithm
  simpleExpr := matchcontinue (expr)
    local
      Exp e,e1,e2,sim1,sim2,res;
      String id;
      list<Exp> exprList,simpleExprList;

    case ADD(e1,e2)
      equation
        sim1 = simplifyExp(e1);
        sim2 = simplifyExp(e2);
        res = simplifyExp2(ADD(sim1,sim2));
      then res;

    case SUB(e1,e2)
      equation
        sim1 = simplifyExp(e1);
```

```
        sim2 = simplifyExp(e2);
        res = simplifyExp2(SUB(sim1,sim2));
      then res;

    case MUL(e1,e2)
      equation
        sim1 = simplifyExp(e1);
        sim2 = simplifyExp(e2);
        res = simplifyExp2(MUL(sim1,sim2));
      then res;

    case DIV(e1,e2)
      equation
        sim1 = simplifyExp(e1);
        sim2 = simplifyExp(e2);
        res = simplifyExp2(DIV(sim1,sim2));
      then res;

    case NEG(e1)
      equation
        sim1 = simplifyExp(e1);
        res = simplifyExp2(NEG(sim1));
      then res;

    case CALL(id,exprList)
      equation
        simpleExprList = simplifyExpList(exprList);
        res = simplifyExp2(CALL(id,simpleExprList));
      then res;

    case e then e; // IDENT, INT
  end matchcontinue;
end simplifyExp;

protected function simplifyExpList
  input list<Exp> exprList;
  output list<Exp> simpleExprList;
algorithm
  simpleExprList := matchcontinue(exprList)
    local
      list<Exp> rest,simpleRest;
      Exp simpleExpr,expr;

    case {} then {};

    case expr::rest
      equation
        simpleExpr = simplifyExp(expr);
        simpleRest = simplifyExpList(rest);
      then simpleExpr :: simpleRest;

  end matchcontinue;
end simplifyExpList;

protected function simplifyExp2
  input Exp expr;
  output Exp simpleExpr;
algorithm
  simpleExpr := matchcontinue (expr)
    local
      Exp e,e1,e2,sim1,sim2;
      Integer i,i1,i2;

    // Simplify addition
    case ADD(INT(i1),INT(i2)) equation i = i1 + i2; then INT(i);
    case ADD(INT(0),e) then e;
```

```
    case ADD(e,INT(0)) then e;

    // Simplify multiplication
    case MUL(INT(i1),INT(i2)) equation i = i1 * i2; then INT(i);
    case MUL(INT(0),e) then INT(0);
    case MUL(e,INT(0)) then INT(0);
    case MUL(INT(1),e) then e;
    case MUL(e,INT(1)) then e;

    // Simplify division
    // case DIV(INT(i1),INT(i2)) can give a real number, so don't simplify
    case DIV(e,INT(1)) then e;

    // Simplify some expressions of these types
    // SUB(INT,INT)
    // SUB(e,0)
    // SUB(0,e)
    // SUB(e1,NEG(e2))

    // NEG(INT)
    // NEG(NEG(e))

    // sin(0)
    // cos(0)
    // pow(x,0)
    // pow(x,1)

    // your code here
    case DIV(e,INT(1)) then e;

    case SUB(INT(i1),INT(i2))
      equation i = i1-i2;
      then INT(i);

    case SUB(INT(0),e1)
      equation
        sim1 = simplifyExp2(NEG(e1));
      then sim1;

    case SUB(e1,NEG(e2))
      equation
        sim2 = simplifyExp2(e2); e = ADD(e1,sim2);
      then
        simplifyExp2(e);

    case NEG(INT(i1))
      equation i = -i1;
      then INT(i);

    case NEG(NEG(e)) then e;
    case CALL("sin",{INT(0)}) then INT(0);
    case CALL("cos",{INT(0)}) then INT(1);
    case CALL("pow",{e,INT(0)}) then INT(1);
    case CALL("pow",{e,INT(1)}) then e;

    // Default case, we can't simplify anymore
    case e then e;
  end matchcontinue;
end simplifyExp2;

// Functions for printing expressions

public function printExp
  input Exp exp;
protected
  String str;
algorithm
```

```
  str := expStr(exp);
  print(str);
end printExp;

protected function expStr
"Translates an Exp into a String"
  input Exp exp;
  output String str;
algorithm
  str := matchcontinue (exp)
    local
      Integer i;
      Exp e,lhs,rhs;
      String left,right,res,id;
      list<Exp> expList;

    case INT(i) then intString(i);

    case ADD(lhs,rhs) then binExpStr(lhs,"+",rhs);
    case SUB(lhs,rhs) then binExpStr(lhs,"-",rhs);
    case MUL(lhs,rhs) then binExpStr(lhs,"*",rhs);
    case DIV(lhs,rhs) then binExpStr(lhs,"/",rhs);
    case NEG(e) then "(-" + expStr(e) + ")";
    case IDENT(id) then id;

    case CALL("der",{e})
      equation
        res = expStr(e);
      then "(" + res + ")'";

    case CALL("pow",{e,INT(i)})
      equation
        res = expStr(e);
      then res + "^" + intString(i);

    case CALL(id,expList)
      equation
        res = expListStr(expList);
      then id + "(" + res + ")";

    case _ then "#UNKNOWN_EXP#";

  end matchcontinue;
end expStr;

protected function expListStr
"Translates a list of Exp into a comma-separated String"
  input list<Exp> expList;
  output String str;
algorithm
  str := matchcontinue (expList)
    local
      Exp e;
      list<Exp> rest;
      String res_1,res_2;

    case {} then "";

    case {e} then expStr(e);

    case e::rest
      equation
        res_1 = expStr(e);
        res_2 = expListStr(rest);
      then res_1 + "," + res_2;
```

```
    end matchcontinue;
  end expListStr;

  protected function binExpStr
  "Translates a binary expression (lhs op rhs) into a String"
    input Exp lhs;
    input String op;
    input Exp rhs;
    output String str;
  algorithm
    str := "(" + expStr(lhs) + op + expStr(rhs) + ")";
  end binExpStr;

  end SymbolicDerivative;
```

E.4 Solution 04 Assignment – Printing AST and Environments

```
  package Assignment "Assignment.mo"

  public
  type ExpLst = list<Exp> "a list of expressions";

  // Abstract syntax for the Assignments language
  uniontype Program "a program"

    record PROGRAM
      ExpLst expLst;
      Exp exp;
    end PROGRAM;

  end Program;

  uniontype Exp "expressions"

    record INT "integer literals"
      Integer integer;
    end INT;

    record BINARY "binary expressions"
      Exp exp1;
      BinOp binOp2;
      Exp exp3;
    end BINARY;

    record UNARY "unary expresions"
      UnOp unOp;
      Exp exp;
    end UNARY;

    record ASSIGN "assignment expressions"
      Ident ident;
      Exp exp;
    end ASSIGN;

    record IDENT "identifiers"
      Ident ident;
    end IDENT;
  end Exp;

  uniontype BinOp "binary operators"
    record ADD "addition"       end ADD;
    record SUB "substraction"   end SUB;
    record MUL "multiplication" end MUL;
    record DIV "division"       end DIV;
  end BinOp;
```

```
uniontype UnOp "unary operators"
  record NEG "negation operator" end NEG;
end UnOp;

type Ident = String;

type Value = Integer "Values stored in environments";

type VarBnd = tuple<Ident,Value> "a binding is a tuple of id and value";

type Env = list<VarBnd> "an environment is a list of bindings";

protected function lookup
 "lookup returns the value associated with an identifier.
  If no association is present, lookup will fail."
  input Env inEnv;
  input Ident inIdent;
  output Value outValue;
algorithm
  outValue := matchcontinue (inEnv,inIdent)
    local
      Ident id2,id;
      Value value;
      Env rest;
    case ((id2,value) :: rest,id)
      then if id == id2
           then value
           else lookup(rest, id);
  end matchcontinue;
end lookup;

protected function lookupextend
  input Env inEnv;
  input Ident inIdent;
  output Env outEnv;
  output Value outValue;
algorithm
  (outEnv,outValue) := matchcontinue (inEnv,inIdent)
    local
      Env env;
      Ident id;
      Value value;

    case (env,id)
      equation
        failure(_ = lookup(env, id));
      then ((id,0) :: env,0);

    case (env,id)
      equation
        value = lookup(env, id);
      then (env,value);
  end matchcontinue;
end lookupextend;

protected function update
  input Env env;
  input Ident id;
  input Value value;
  output Env outEnv;
algorithm
  outEnv := (id,value) :: env;
end update;

protected function applyBinop
  input BinOp inBinOp1;
  input Integer inInteger2;
```

```
  input Integer inInteger3;
  output Integer outInteger;
algorithm
  outInteger := matchcontinue (inBinOp1,inInteger2,inInteger3)
    local Value v1,v2;
    case (ADD(),v1,v2) then v1+v2;
    case (SUB(),v1,v2) then v1-v2;
    case (MUL(),v1,v2) then v1*v2;
    case (DIV(),v1,v2) then intDiv(v1,v2);
  end matchcontinue;
end applyBinop;

protected function applyUnop
  input UnOp inUnOp;
  input Integer inInteger;
  output Integer outInteger;
algorithm
  outInteger := match (inUnOp,inInteger)
    local Value v;
    case (NEG(),v) then -v;
  end match;
end applyUnop;

protected function eval
  input Env inEnv;
  input Exp inExp;
  output Env outEnv;
  output Integer outInteger;
algorithm
  (outEnv,outInteger) := matchcontinue (inEnv,inExp)
    local
      Env env,env2,env3,env1;
      Value ival,value,v1,v2,v3;
      Ident s,id;
      Exp exp,e1,e2,e;
      BinOp binop;
      UnOp unop;

    // eval of integer record returns the integer value
    case (env,INT(integer = ival)) then (env,ival);

    // eval of an identifier node will lookup the identifier and return a
    // value if present; otherwise insert a binding to zero, and return zero.
    case (env,IDENT(ident = id))
      equation
        (env2,value) = lookupextend(env, id);
      then (env2,value);

    // eval of an assignment node returns the
    // updated environment and the assigned value.
    case (env,ASSIGN(ident = id,exp = exp))
      equation
        (env2,value) = eval(env, exp);
        env3 = update(env2, id, value);
      then (env3,value);

    // eval of a node e1,ADD,e2 , etc. in an environment env
    case (env1,BINARY(exp1 = e1,binOp2 = binop,exp3 = e2))
      equation
        (env2,v1) = eval(env1, e1);
        (env3,v2) = eval(env2, e2);
        v3 = applyBinop(binop, v1, v2);
      then (env3,v3);

    // eval of a node NEG,e etc. in an environment env
    case (env1,UNARY(unOp = unop,exp = e))
      equation
```

```
        (env2,v1) = eval(env1, e);
        v2 = applyUnop(unop, v1);
      then (env2,v2);

  end matchcontinue;
end eval;

protected function evals
  input Env inEnv;
  input ExpLst inExpLst;
  output Env outEnv;
algorithm
  outEnv := matchcontinue (inEnv,inExpLst)
    local
      Env e,env2,env3,env;
      Value v;
      Ident s;
      Exp exp;
      ExpLst expl;

    // the environment stay the same if there are no expressions
    case (e,{}) then e;

    // the head expression is evaluated in the current environment
    // generating a new environment in which the rest of the expression
    // list is evaluated. the last environment is returned
    case (env,exp :: expl)
      equation
        (env2,v) = eval(env, exp);
        env3 = evals(env2, expl);
      then env3;

  end matchcontinue;
end evals;

public function evalprogram
  input Program inProgram;
  output Integer outInteger;
algorithm
  outInteger := match (inProgram)
    local
      ExpLst assignments_1,assignments;
      Env env2;
      Value value;
      Exp exp;

    case (PROGRAM(expLst = assignments,exp = exp))
      equation
        assignments_1 = listReverse(assignments);
        print("\nThe assignments: ");
        printAssignments(assignments_1);
        print("The expression: ");
        printExp(exp);
        env2 = evals({}, assignments_1);
        print("\nThe environment: ");
        printEnvironment(env2);
        (_,value) = eval(env2, exp);
      then value;

  end match;
end evalprogram;

function printAssignments
  input ExpLst assignList;
algorithm
  _ := matchcontinue(assignList)
```

```
    local ExpLst expLst; Exp exp;

    case ({}) // if nothing is in the list don't print anything
      then ();

    case (exp::{})
      equation
        printExp(exp);
        print("\n");
      then ();

    case (exp::expLst)
      equation
        printExp(exp);
        print(", ");
        printAssignments(expLst);
      then ();

  end matchcontinue;
end printAssignments;


function printExp
  input Exp inExp;
algorithm
  _ := matchcontinue(inExp)
    local
      Integer i;
      Exp exp1, exp2, exp;
      Ident id;
      BinOp binop;
      UnOp unop;

    case(INT(i))
      equation
        print(intString(i));
      then ();

    case(BINARY(exp1, binop, exp2))
      equation
        printExp(exp1);
        printBinaryOp(binop);
        printExp(exp2);
      then ();

    case (UNARY(unop, exp))
      equation
        printUnaryOp(unop);
        printExp(exp);
      then ();

    case(ASSIGN(id, exp))
      equation
        print(id);
        print(" = ");
        printExp(exp);
      then ();

    case(IDENT(id))
      equation
        print(id);
      then ();

    case _ // handle failure
      equation
        print("- printExp failed\n");
      then fail();
```

```
  end matchcontinue;
end printExp;

function printBinaryOp
  input BinOp op;
algorithm
  _ := matchcontinue (op)
    case (ADD()) equation print("+"); then ();
    case (SUB()) equation print("-"); then ();
    case (MUL()) equation print("*"); then ();
    case (DIV()) equation print("/"); then ();
  end matchcontinue;
end printBinaryOp;

function printUnaryOp
  input UnOp op;
algorithm
  _ := match (op)
    case (NEG())
      equation print("-");
      then ();
  end match;
end printUnaryOp;

function printEnvironment
  input Env varBndList;
algorithm
  _ := matchcontinue(varBndList)
    local
      Ident id;
      Value val;
      Env varBndLstRest;

    case ({}) then (); // if nothing is in the list don't print anything

    case ((id, val)::{}) // the last id and value
      equation
        print(id);
        print(" = ");
        print(intString(val));
        print("\n");
      then ();

    case ((id, val)::varBndLstRest) // the usual case
      equation
        print(id);
        print(" = ");
        print(intString(val));
        print(", ");
        printEnvironment(varBndLstRest);
      then ();

  end matchcontinue;
end printEnvironment;

end Assignment;
```

E.5 Solution 05a AssignTwoType – Adding a New Type to a Language

```
package AssignTwoType "file AssignTwoType.mo"

public
type ExpLst = list<Exp> "expression list";

// Abstract syntax for the Assigntwotype language
```

```
uniontype Program "a program"
  record PROGRAM
    ExpLst expLst;
    Exp exp;
  end PROGRAM;
end Program;

public
uniontype Exp "expressions"
  record INT "literal integers"
    Integer integer;
  end INT;

  record REAL "literal reals"
    Real real;
  end REAL;

  record STRING "literal strings"
    String string;
  end STRING;

  record BINARY "binary expressions"
    Exp exp1;
    BinOp binOp2;
    Exp exp3;
  end BINARY;

  record UNARY "unary expressions"
    UnOp unOp;
    Exp exp;
  end UNARY;

  record ASSIGN "assignment expressions"
    Ident ident;
    Exp exp;
  end ASSIGN;

  record IDENT "identifiers"
    Ident ident;
  end IDENT;

end Exp;

public
uniontype BinOp "binary operators"
  record ADD "addition operator"       end ADD;
  record SUB "subtraction operator"    end SUB;
  record MUL "multiplication operator" end MUL;
  record DIV "divition operator"       end DIV;
end BinOp;

public
uniontype UnOp "unary operators"
  record NEG "negation operator" end NEG;
end UnOp;

public
type Ident = String;

public
uniontype Value "Values stored in environments"
  record INTval "integer values"
    Integer integer;
  end INTval;

  record REALval "real values"
    Real real;
```

```
  end REALval;
end Value;

public
type VarBnd = tuple<Ident,Value> "Bindings and environments";

public
type Env = list<VarBnd>;

public
uniontype Ty2 "Ty2 is an auxiliary datatype used to
               handle types during evaluation"

  record INT2
    Integer integer1;
    Integer integer2;
  end INT2;

  record REAL2
    Real real1;
    Real real2;
  end REAL2;

end Ty2;

protected function printvalue
  input Value inValue;
algorithm
  _ := matchcontinue (inValue)
    local
      Ident str;
      Integer i;
      Real r;

    case (INTval(integer = i))
      equation
        str = intString(i);
        print(str);
      then ();

    case (REALval(real = r))
      equation
        str = realString(r);
        print(str);
      then ();

  end matchcontinue;
end printvalue;

public function evalprogram
  input Program inProgram;
algorithm
  _ := matchcontinue (inProgram)
    local
      ExpLst assignments_1,assignments;
      Env env2;
      Value value;
      Exp exp;

    case (PROGRAM(expLst = assignments,exp = exp))
      equation
        assignments_1 = listReverse(assignments);
        env2 = evals({}, assignments_1);
        (_,value) = eval(env2, exp);
        printvalue(value);
        print("\n");
```

```
      then ();

  end matchcontinue;
end evalprogram;

protected function evals
  input Env inEnv;
  input ExpLst inExpLst;
  output Env outEnv;
algorithm
  outEnv := matchcontinue (inEnv,inExpLst)
    local
      Env e,env2,env3,env;
      Exp exp;
      ExpLst expl;

    case (e,{}) then e;

    case (env,exp :: expl)
      equation
        (env2,_) = eval(env, exp);
        env3 = evals(env2, expl);
      then env3;

  end matchcontinue;
end evals;

protected function eval
  input Env inEnv;
  input Exp inExp;
  output Env outEnv;
  output Value outValue;
algorithm
  (outEnv,outValue) := matchcontinue (inEnv,inExp)
    local
      Env env,env2,env1;
      Integer ival,x,y,z;
      Real rval,rx,ry,rz;
      String sval;
      Value value,v1,v2;
      Ident id;
      Exp e1,e2,e,exp;
      BinOp binop;
      UnOp unop;

    // handle int
    case (env,INT(integer = ival)) then (env,INTval(ival));

    // handle real
    case (env,REAL(real = rval)) then (env,REALval(rval));

    // handle string
    case (env,STRING(string = sval))
      equation
        ival = stringInt(sval);
      then (env,INTval(ival));

    // variable id
    case (env,IDENT(ident = id))
      equation
        (env2,value) = lookupextend(env, id);
      then (env2,value);

    // int binop int
    case (env,BINARY(exp1 = e1,binOp2 = binop,exp3 = e2))
      equation
        (env1,v1) = eval(env, e1);
```

```
        (env2,v2) = eval(env, e2);
        INT2(integer1 = x,integer2 = y) = typeLub(v1, v2);
        z = applyIntBinop(binop, x, y);
      then (env2,INTval(z));

    // int/real binop int/real
    case (env,BINARY(exp1 = e1,binOp2 = binop,exp3 = e2))
      equation
        (env1,v1) = eval(env, e1);
        (env2,v2) = eval(env, e2);
        REAL2(real1 = rx,real2 = ry) = typeLub(v1, v2);
        rz = applyRealBinop(binop, rx, ry);
      then (env2,REALval(rz));

    // int unop exp
    case (env,UNARY(unOp = unop,exp = e))
      equation
        (env1,INTval(integer = x)) = eval(env, e);
        y = applyIntUnop(unop, x);
      then (env1,INTval(y));

    // real unop exp
    case (env,UNARY(unOp = unop,exp = e))
      equation
        (env1,REALval(real = rx)) = eval(env, e);
        ry = applyRealUnop(unop, rx);
      then (env1,REALval(ry));

    // eval of an assignment node returns the updated
    // environment and the assigned value id := exp
    case (env,ASSIGN(ident = id,exp = exp))
      equation
        (env1,value) = eval(env, exp);
        env2 = update(env1, id, value);
      then (env2,value);

  end matchcontinue;
end eval;

protected function typeLub
  input Value inValue1;
  input Value inValue2;
  output Ty2 outTy2;
algorithm
  outTy2 := matchcontinue (inValue1,inValue2)
    local
      Integer x,y;
      Real x2,y2;

    case (INTval(integer = x),INTval(integer = y)) then INT2(x,y);

    case (INTval(integer = x),REALval(real = y2))
      equation
        x2 = intReal(x);
      then REAL2(x2,y2);

    case (REALval(real = x2),INTval(integer = y))
      equation
        y2 = intReal(y);
      then REAL2(x2,y2);

    case (REALval(real = x2),REALval(real = y2))
      then REAL2(x2,y2);
  end matchcontinue;
end typeLub;

protected function applyIntBinop
```

```
    input BinOp inBinOp1;
    input Integer inInteger2;
    input Integer inInteger3;
    output Integer outInteger;
  algorithm
    outInteger := matchcontinue (inBinOp1,inInteger2,inInteger3)
      local Integer x,y;
      case (ADD(),x,y) then x + y;
      case (SUB(),x,y) then x - y;
      case (MUL(),x,y) then x * y;
      case (DIV(),x,y) then intDiv(x, y);
    end matchcontinue;
  end applyIntBinop;

  protected function applyRealBinop
    input BinOp inBinOp1;
    input Real inReal2;
    input Real inReal3;
    output Real outReal;
  algorithm
    outReal := matchcontinue (inBinOp1,inReal2,inReal3)
      local Real x,y;
      case (ADD(),x,y) then x + y;
      case (SUB(),x,y) then x - y;
      case (MUL(),x,y) then x * y;
      case (DIV(),x,y) then x / y;
    end matchcontinue;
  end applyRealBinop;

  protected function applyIntUnop
    input UnOp inUnOp;
    input Integer inInteger;
    output Integer outInteger;
  algorithm
    outInteger := matchcontinue (inUnOp,inInteger)
      local Integer x;
      case (NEG(),x) then -x;
    end matchcontinue;
  end applyIntUnop;

  protected function applyRealUnop
    input UnOp inUnOp;
    input Real inReal;
    output Real outReal;
  algorithm
    outReal := matchcontinue (inUnOp,inReal)
      local Real x;
      case (NEG(),x) then -. x;
    end matchcontinue;
  end applyRealUnop;

  protected function lookup
    input Env inEnv;
    input Ident inIdent;
    output Value outValue;
  algorithm
    outValue := matchcontinue (inEnv,inIdent)
      local
        Ident id2,id;
        Value value;
        Env rest;

      // lookup returns the value associated with an identifier.
      // If no association is present, lookup will fail.
      // Identifier id is found in the first pair of the list,
      // and value is returned.
      case ((id2,value) :: _,id)
```

```
        equation
          equality(id = id2);
        then value;

      // id is not found in the first pair of the list, and lookup will
      // recursively search the rest of the list. If found, value is returned.
      case ((id2,_) :: rest,id)
        equation
          failure(equality(id = id2));
          value = lookup(rest, id);
        then value;

    end matchcontinue;
  end lookup;

  protected function lookupextend
    input Env inEnv;
    input Ident inIdent;
    output Env outEnv;
    output Value outValue;
  algorithm
    (outEnv,outValue) := matchcontinue (inEnv,inIdent)
      local
        Value value;
        Env env;
        Ident id;
      // Return value of id in env. If id not present, add id and return 0
      case (env,id)
        equation
          failure(value = lookup(env, id)); // failed to find id
          value = INTval(0);
        then ((id,value) :: env,value);

      case (env,id)
        equation
          value = lookup(env, id); // found id, return it
        then (env,value);

    end matchcontinue;
  end lookupextend;

  protected function update
    input Env inEnv;
    input Ident inIdent;
    input Value inValue;
    output Env outEnv;
  algorithm
    outEnv := matchcontinue (inEnv,inIdent,inValue)
      local
        Env env;
        Ident id;
        Value value;
      case (env,id,value) then (id,value) :: env;
    end matchcontinue;
  end update;

  end AssignTwoType;
```

E.6 Solution 05b ModAssignTwoType – Adding a New Type to a Language

Analogous to 05a AssignTwoType.

E.7 Solution 06 Advanced – Polymorphic Types and Higher Order Functions

```
package Main

import Types;
import Functions;

function main
  input list<String> arg;
algorithm
  _ := matchcontinue arg
    local
      list<Integer> orderedIntList;
      list<String> orderedStringList, stringList, strRealLst, strIntLst;
      list<Real> orderedRealList, realList;
    case (_)
      equation
        // your code here:
        // order the initial Int list
        orderedIntList =
          Functions.orderList(Types.intList, Functions.compareInt);
        // transform the ordered list to String for printing
        strIntLst =
          Functions.map1(orderedIntList, Functions.transformInt2String);
        print("Int String List:");
        Functions.map0(strIntLst, Functions.printElement);

        // transforming the initial int list to a String list
        stringList =
          Functions.map1(Types.intList, Functions.transformInt2String);
        // order the transformed String list
        orderedStringList =
          Functions.orderList(stringList, Functions.compareString);
        print("\nOrdered String List:");
        Functions.map0(orderedStringList, Functions.printElement);

        // transforming the int list to a Real list
        realList = Functions.map1(Types.intList, Functions.transformInt2Real);
        // order the transformed Real list
        orderedRealList = Functions.orderList(realList, Functions.compareReal);

        strRealLst =
          Functions.map1(orderedRealList, Functions.transformReal2String);
        print("\nOrdered Real List:");
        Functions.map0(strIntLst, Functions.printElement);
        print("\n");
      then ();
  end matchcontinue;
end main;

end Main;

package Functions

function compareInt
  input Integer i1;
  input Integer i2;
  output Boolean b;
algorithm
  b := i1 < i2;
end compareInt;

function compareReal
  input Real r1;
  input Real r2;
```

```
  output Boolean b;
algorithm
  b := r1 <. r2;
end compareReal;

function compareString
  input String s1;
  input String s2;
  output Boolean b;
algorithm
  b := matchcontinue (s1, s2)
    local Integer z;
    case (s1, s2)
      equation
        0 = stringCompare(s1, s2);
      then false;
    case (s1, s2)
      equation
        z = stringCompare(s1, s2);
        true = (z < 0);
      then true;
    case (s1, s2)
      equation
        z = stringCompare(s1, s2);
        false = (z < 0);
      then false;
  end matchcontinue;
end compareString;

function quicksort<Type_a>
  input  list<Type_a> inList;
  input  list<Type_a> accList;
  input FuncType comparator;
  output list<Type_a> outList;
public
  partial function FuncType
    input Type_a el1;
    input Type_a el2;
    output Boolean cmp;
  end FuncType;
algorithm
  outList := matchcontinue (inList, accList, comparator)
    local
      list<Type_a> l, smaller, greater, acc, lst1, lst2, lst3;
      Type_a x;
    case ({}, acc, _) then acc;  // handle empty case
    case (x::l, acc, comparator) // handle the reminder
      equation
        (smaller, greater) = partition (x, l, comparator);
        lst1 = quicksort (greater, acc, comparator);
        lst2 = x::lst1;
        lst3 = quicksort (smaller, lst2, comparator);
      then lst3;
  end matchcontinue;
end quicksort;

function partition<Type_a>
  input  Type_a inList;
  input  list<Type_a> accList;
  input FuncType comparator;
  output list<Type_a> outList1;
  output list<Type_a> outList2;
public
  partial function FuncType
    input Type_a el1;
    input Type_a el2;
    output Boolean cmp;
```

```
    end FuncType;
  algorithm
    (outList1,outList2) := matchcontinue (inList, accList, comparator)
      local
        Type_a x, y;
        list<Type_a> l, smaller, greater;

      case (x, {}, _) then ({}, {});

      case (x, y::l, comparator)
        equation
          (smaller, greater) = partition (x, l, comparator);
          true = comparator(y, x);
        then (y::smaller, greater);

      case (x, y::l, comparator)
        equation
          (smaller, greater) = partition (x, l, comparator);
          false = comparator(y, x);
        then (smaller, y::greater);

    end matchcontinue;
  end partition;

  function orderList<Type_a>
    input  list<Type_a> inList;
    input FuncType comparator;
    output list<Type_a> outList;
  public
    partial function FuncType
      input Type_a el1;
      input Type_a el2;
      output Boolean cmp;
    end FuncType;
  algorithm
    outList := matchcontinue (inList, comparator)
      local list<Type_a> lst, lstResult;

      case ({}, _) then {};

      case (lst, comparator)
        equation
          lstResult = quicksort(lst, {}, comparator);
        then lstResult;

    end matchcontinue;
  end orderList;

  // transformer functions
  function transformInt2Real
    input Integer i;
    output Real r;
  algorithm
    r := intReal(i);
  end transformInt2Real;

  function transformInt2String
    input Integer i;
    output String s;
  algorithm
    s := intString(i);
  end transformInt2String;

  function transformReal2String
    input Real r;
    output String s;
  algorithm
```

```
  s := realString(r);
end transformReal2String;

// mapping functions
function map1<Type_a,Type_b>
  input  list<Type_a> inList;
  input FuncType f;
  output list<Type_b> outList;
public
  partial function FuncType
    input  Type_a elIn;
    output Type_b elOut;
  end FuncType;
algorithm
  outList := matchcontinue(inList, f)
    local
      list<Type_b> lst;
      list<Type_a> rest;
      Type_a x;
      Type_b y;

    case ({},_) then {}; // handle empty case

    case (x::rest, f)    // handle the usual case
      equation
        y = f(x);
        lst = map1(rest, f);
      then y::lst;
  end matchcontinue;
end map1;

function map0<Type_a>
  input  list<Type_a> inList;
  input FuncType f;
public
  partial function FuncType
    input  Type_a elIn;
  end FuncType;
algorithm
  _ := matchcontinue(inList, f)
    local
      list<Type_a> rest;
      Type_a x;

    case ({},_) then ();

    case (x::rest, f)
      equation
        f(x);
        map0(rest, f);
      then ();
  end matchcontinue;
end map0;

function printElement
  input String str;
algorithm
  print(str);
  print(" ");
end printElement;

end Functions;
```

E.8 Solution 07_pam – A small Language

The solution is available in the files Input.mo, Main.mo, Parse.mo, Pam.mo, gram.y, lexer.l and in Appendix C.1.

E.9 Solution 08_pamdecl – Pam with Declarations

The solution is available in the files Absyn.mo, Eval.mo, Env.mo, Main.mo, ScanParse.mo, gram.y and in Appendix C.

E.10 Solution 09_pamtrans – Small Translational Semantics

The solution is available in the files Absyn.mo, Emit.mo, Main.mo, Mcode.mo, Parse.mo, Trans.mo, gram.y, lexer.l and in Appendix C.

E.11 Solution 10_Petrol – Large Translational Semantics

The solution is available in the files Absyn.mo, FCEmit.mo, FCode.mo, Flatten.mo, Main.mo, Parse.mo, Static.mo, TCode.mo, Types.mo, Parser.y, lexer.c.

References

ACE – Associated Computer Experts. The CoSy Compiler Generation System. www.ace.nl and www.opencosy.org. Last Accessed January 2011.

Martin Alt. *On Parallel Compilation*. PhD thesis, University of Saarbrücken, 1997.

Niclas Andersson, Peter Fritzson. Overview and Industrial Application of Code Generator Generators. *Journal of Systems and Software*, Vol 32, No 3, pp 185-214, March 1996.

Peter Aronsson, Peter Fritzson, Levon Saldamli, Peter Bunus and Kaj Nyström. Meta Programming and Function Overloading in OpenModelica. In *Proceedings of the 3rd International Modelica Conference*, Linköping, Sweden, Nov 2003.

Uwe Assmann. Graph Rewrite Systems for Program Optimization. *ACM Transactions on Programming Languages and Systems* (TOPLAS), Volume 22 Issue 4, July 2000.

Patrik Borras, Dominique Clement, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. CENTAUR: The System. In *Proc. of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. No 24 of SIGPLAN Notices pp. 14-24. 1988.

David Broman and Peter Fritzson. Abstract Syntax Can Make the Definition of Modelica Less Abstract. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools, (EOOLT'2007)*, Berlin, July 30, 2007. Published by Linköping University Electronic Press, http://www.ep.liu.se/ecp/024/, July 2007.

David Broman and Peter Fritzson. Higher-Order Acausal Models. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools, (EOOLT'2008*), Pathos, Cyprus, July 8, 2008. Published by Linköping University Electronic Press, http://www.ep.liu.se/ecp/024/, July 2008.

David Broman. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation*. Dissertation No 1333, www.ep.liu.se, Linköping University, October 1, 2010.

Stefan Brus. Bootstrapping The OpenModelica Compiler: Implementing Functions As Arguments. Master thesis draft, 2010. www.ep.liu.se. Finalized Spring 2011.

Emil Carlsson. Translating Natural Semantics to Meta-Modelica. Master Thesis, LITH-IDA-Ex--05/073—SE, Linköping University, October 2005.

Thierry Despeyroux. Executable Specification of Static Semantics. In *Semantics of Data Types*. 1984. Berlin, Germany. Springer-Verlag. Lecture Notes in Computer Science (LNCS) No:173. pp. 215-23. 1984.

Thierry Despeyroux. TYPOL: A Formalism to Implement Natural Semantics. 1988, INRIA, Sofia-Antipolis. http://www.inria.fr/rrrt/rt-0094.html. 1988.

Helmut Emmelmann, F. W. Schröer, Rudolf Landwehr. BEG – A Generator for Efficient Back Ends. *ACM Sigplan Notices*, Vol 24, No 7, pp 227-237, 1989.

Dawson R. Engler and Massimiliano Poletto. *A 'C Tutorial*, Technical Report, 1997.

Christopher Fraser and David Hansen. *A Retargetable C Compiler: Design and Implementation*. ISBN 10: 0805316701, Addison-Wesley, 1995.

Peter Fritzson. *Efficient Language Implementation by Natural Semantics, Generating Efficient Language Implementations from Operational Semantics* (2008). 1996-2008, several similar draft versions. www: http://www.ida.liu.se/~pelab/rml. Last Accessed: 2008. 1996.

Peter Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, 940 pages, Wiley-IEEE Press, 2004.

Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe*, 44/45, Dec. 2005. http://www.openmodelica.org

Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proc. of the 4th International Modelica Conference*, Hamburg, Germany, March 7-8, 2005.

Peter Fritzson. *Language Modeling and Symbolic Transformations with Meta-Modelica*. (Later versions 2007 and 2011), www.ida.liu.se/~pelab/Modelica, and www.openmodelica.org Version 0.5, June 2005.

Peter Fritzson. *Modelica Meta-Programming and Symbolic Transformations - MetaModelica Programming Guide*. (Slightly updated compared to v0.5 2005) http://www.openmodelica.org/index.php/developer/devdocumentation. June 2007.

Peter Fritzson, Adrian Pop, David Broman, Peter Aronsson. Formal Semantics Based Translator Generation and Tool Development in Practice. In *Proceedings of ASWEC 2009 Australian Software Engineering Conference*, Gold Coast, Australia, April, 2009.

Peter Fritzson, Pavol Privitzer, Martin Sjölund, and Adrian Pop. Towards a text generation template language for Modelica. In *Proceedings of the 7th International Modelica Conference*, Como, Italy, Sept. 20-22, 2009

Peter Fritzson and Adrian Pop. Meta-Programming and Language Modeling with MetaModelica 1.0. Technical reports in Computer and Information Science, No 9, Linköping University Electronic Press, http://www.ep.liu.se/PubList/Default.aspx?SeriesID=2550, March 2011.

Peter Fritzson, Adrian Pop, Martin Sjölund, Per Östlund, Peter Aronsson, David Akhvlediani, Adeel Asghar, Bernhard Bachmann, Vasile Baluta,Simon Björklén, Mikael Blom, Robert Braun, Willi Braun, David Broman, Stefan Brus, Francesco Casella, Filippo Donida, Henrik Eriksson, Anders Fernström, Jens Frenkel, Pavel Grozman, Daniel Hedberg, Michael Hanke, Zoheb Hossain, Alf Isaksson, Kim Jansson, Daniel Kanth, Tommi Karhela, Joel Klinghed, Juha Kortelainen, Petter Krus, Alexey Lebedev, Magnus Leksell, Oliver Lenord, Ariel Liebman, Rickard Lindberg, Håkan Lundvall, Henrik Magnusson, Eric Meyers, Hannu Niemistö, Peter Nordin, Kristoffer Norling, Lennart Ochel, Atanas Pavlov, Karl Pettersson, Pavol Privitzer, Reino Ruusu, Per Sahlin, Ingo Staack, Wladimir Schamai, Gerhard Schmitz, Klas Sjöholm, Anton Sodja, Kristian Stavåker, Sonia Tariq, Mohsen Torabzadeh-Tari, Parham Vasaiely, Niklas Worschech, Robert Wotzlaw, Björn Zackrisson, Azam Zia. *OpenModelica Users Guide, version 1.7*. www.openmodelica.org. March 2011.

Samuel Harbison and Guy Steele. *C – A Reference Manual*. Prentice Hall, 1984.

J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29-60, Dec. 1969.

David Kågedal. A Natural Semantics specification for the equation-based modeling language Modelica. Master Thesis, Department of Computer and Information Science, Linköping University, 1998.

Rudolf Landwehr, H.S. Jansohn, Gerhard Goos. Experience with an Automatic Code Generator Generator. *ACM Sigplan Notices*, Vol 17, No 6, pp 56-66, 1982.

Kenneth C. Louden. *Programming Languages, Principles and Practice*. ISBN 0-534-95341-7, Thomson Brooks/Cole, 2003.

Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:248-375, 1978.

Robin Milner, Mads Tofte, Robert Harper and David MacQueen. *The Definition of Standard ML*, 128 pages, MIT Press, 1997.

Modelica Association. *The Modelica Language Specification Version 3.2*, March 2010. http://www.modelica.org.

Peter D. Mosses, Modular structural operational semantics. *Journal of Functional Programming and Algebraic Programming*. Special issue on SOS., No 60-61: pp. 195-22, 2004.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Available online: http://www.itu.dk/people/sestoft/pebook/, Prentice Hall International, ISBN 0-13-020249-5, June 1993.

Peter Fritzson and David Kågedal. Generating a Modelica Compiler from Natural Semantics Specifications. In *Proc. of 1998 Summer Computer Simulation Conference (SCSC'98)*, Reno, Nevada. 1998.

Henrik Nilsson, John Peterson, and Paul Hudak. Functional Hybrid Modeling from an Object-Oriented Perspective. In *Proc. of EOOLT'2007*, LIU Electronic Press, www.ep.liu.se, Berlin, Germany, August, 2007.

Ulf Nilsson and Jan Maluszynski. *Logic, Programming and Prolog*. Wiley, 1995.

Objective Caml 3.12.0 installation notes. Aug, 2010. http://caml.inria.fr

Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, 2008.

Frank Pagan. *Formal Specification of Programming Languages A Panoramic Primer*. ISBN 0-13-329052-2, Prentice Hall, 1981.

Terence Parr. Enforcing Strict Model-View Separation in Template Engines. http://www.stringtemplate .org,. May 2004. (Accessed May 2009).

Terence Parr. [DRAFT] A Functional Language For Generating Structured Text. http://www.stringtemplate.org. May 2006. (Accessed May 2009).

Terence Parr. StringTemplate documentation. http://www.stringtemplate.org. Accessed May 2009.

Mikael Pettersson. *Compiling Natural Semantics*, Department of Computer and Information Science, Linköping University, PhD Thesis No. 413, 1995. Published in *Lecture Notes in Computer Science* No 1549, Springer Verlag, 1999.

Benjamin Pierce. *Types and Programming Languages*. ISBN 0-262-16209-1. MIT Press, 2002.

Gordon Plotkin. A structural approach to operational semantics. Århus University, Århus, Denmark, 1981.

Adrian Pop, Peter Fritzson, Andreas Remar, Elmir Jagudin, and David Akhvlediani. OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging. In *Proc 5th International Modelica Conf. (Modelica'2006)*, Vienna, Austria, Sept. 4-5, 2006.

Adrian Pop and Peter Fritzson. MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language. In *Proceedings of Joint Modular Languages Conference 2006* (JMLC2006) Published in Lecture Notes in Computer Science No 4228, ISSN 0302-9743, Springer Verlag. Jesus College, Oxford, England, Sept 13-15, 2006.

Adrian Pop, Kristian Stavåker, and Peter Fritzson. Exception Handling for Modelica. In *Proceedings of the 6th International Modelica Conference* (Modelica'2008), Bielefeld, Germany, March.3-4, 2008.

Adrian Pop. *Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages*. www.ep.liu.se, PhD Thesis No. 1183, Linköping University, June 5, 2008.

Tim Sheard. Accomplishments and Research Challenges in Meta-Programming. *Lecture Notes in Computer Science*, 2196, 2001.

Martin Sjölund. Bidirectional External Function Interface Between Modelica/MetaModelica and Java. Master thesis, IDA/LITHEXA09/041SE, www.ep.liu.se, Aug 2009.

Kristian Stavåker, Adrian Pop, and Peter Fritzson. Compiling and Using Pattern Matching in Modelica. In *Proceedings of the 6th International Modelica Conference (Modelica'2008)*, Bielefeld, Germany, March.3-4, 2008.

Guy Steele Jr., *Common Lisp the Language*, 2nd Edition, pp 1029, Digital Press, ISBN 1-55558-041-6, On-line version http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/clm.html, 1990.

Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997, pages 203–217. New York: ACM, 1997.

Wikipedia Partial Evaluation. http://en.wikipedia.org/wiki/Partial_evaluation, Last Accessed January 2011.

Wikipedia OCAML. http://en.wikipedia.org/wiki/Objective_Caml#References, Last Accessed March 2011.

StephenWolfram. *The Mathematica Book*. 5th Ed. Wolfram Media, Inc, 2003.

Index

and, 183
built-in
  operators, 98
constant, 97
else, 183
floating point arithmetic, 99
fully qualified name, 147
function declaration, 135
if, 183
keyword
  constant, 97
  else, 183
  if, 183
  then, 183
keywords, 97
literals
  list, 105
not, 183
operators, 183
  *, 183
  ., 183
  /, 183
  [ ], 183
  +, 183
  <, 183
  <=, 183
  ==, 183
  >, 183
  >=, 183
  and, 183
  not, 183
  or, 183
or, 183
predefined types, 97
then, 183