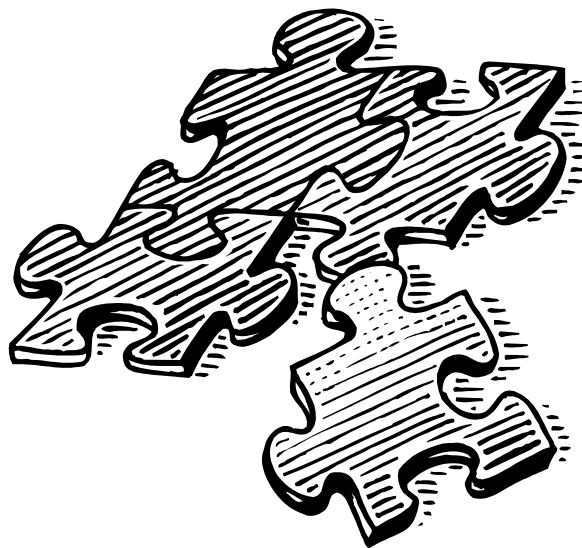# Meta-Programming and Language Modeling with MetaModelica 1.0

Version February 2011

2011-02-28

Peter Fritzson and Adrian Pop

Table of Contents

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, 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 smaller languages: both imperative, functional, and object-oriented. 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-driven requirements on the application 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 1.0. This 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 described in this report has been in extensive use during 2005-2011, primarily for development of the OpenModelica compiler. This has been successful, and has resulted in an efficient and portable OpenModelica implementation. However, the MetaModelica 1.0 language has the drawback of not supporting many features in the standard Modelica language.

The next version of MetaModelica, called MetaModelica 2.0 is described in a separate report. This language is easier to use for a person who knows Modelica since it supports the standard Modelica 3 language features, most of the MetaModelica 1.0 features, as well as additional modeling features for expressiveness and conciseness. It is implemented as part of the OpenModelica compiler itself and is not dependent on the old RML compiler kernel. MetaModelica 2.0 is becoming operational during spring 2011.

Linköping, Sweden, February 2011

Peter Fritzson and Adrian Pop

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.

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 increase, 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, makes 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, that 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 (Plotking 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 this text is on the topic of generating compilers and interpreters from specifications in MetaModelica.

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, and have therefore chosen this formalism for semantics specification in this text.
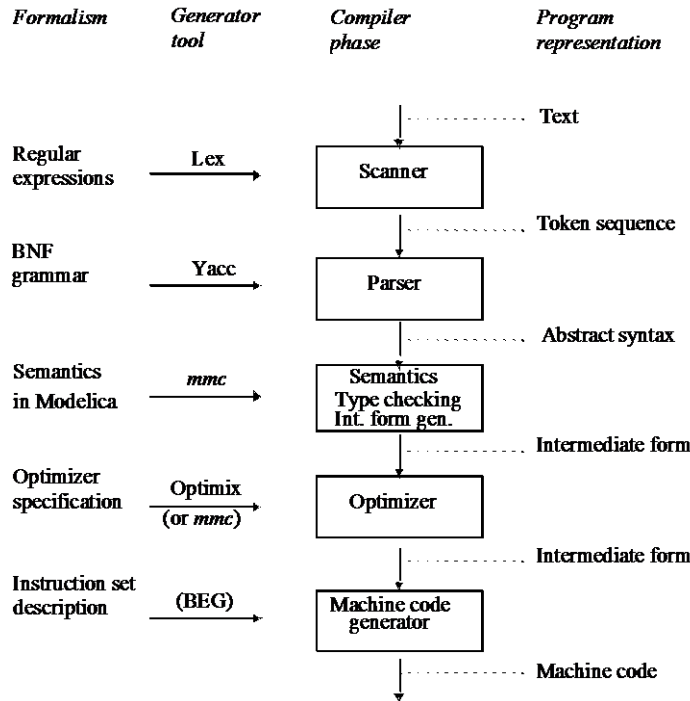
Figure 1-1. Generation of implementations of compiler phases from different formalisms. MetaModelica is used to specify the semantics module, which is generated using the mmc tool (MetaModelica Compiler).

The second necessary requirement for widespread practical use of automatic generation of semantics parts of language implementations is that the generated result need to be roughly as efficient as hand-written implementations., a generator tool, `mmc` (MetaModelica 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.
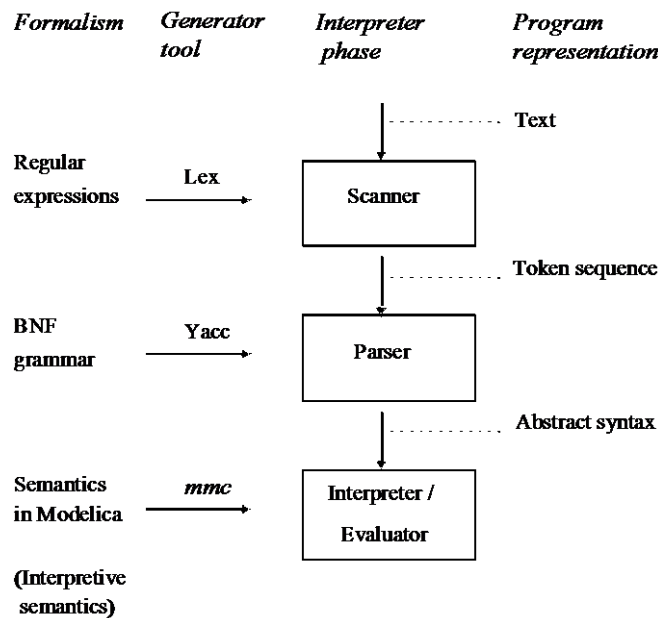
Figure 1-2. Generation of a typical interpreter. The program text is converted into an abstract syntax representation, which is then evaluated by an interpreter generated by the MetaModelica Compiler (mmc) system. Alternatively, some other intermediate representation such as postfix code can be produced, 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.

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 automatically imported into the surrounding scope.
- 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`, `list`, `tuple`, and `Option`.

```
                          ADDop
                    ╱               ╲
              INTconst              MULop
                 │                ╱        ╲
                12          INTconst      INTconst
                                │             │
                                5            13
```
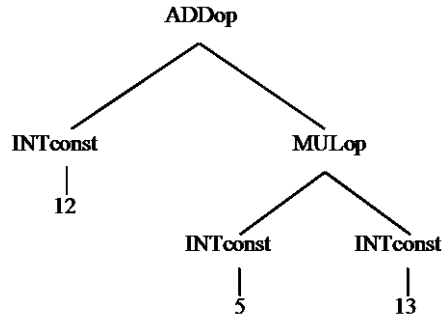
Figure 2-1. Abstract syntax tree of 12+5*13 in the language Exp1.

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.

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 consists 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 `mathcontinue` 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 :=
     matchcontinue s
       case "one"   then 1;
       case "two"   then 2;
       case "three" then 3;
       case _       then 0;
     end matchcontinue;
```

Alternatively, using `match` instead of `matchcontinue`:

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

Match-expressions have the following properties (see Section 5.16.1.2 for a more precise description):

- `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.

- `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*.

- 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.

- Equations are solved in the order they are declared (this restriction may be removed in the future).

- 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 `mathcontinue` 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 := matchcontinue inExp
    local Integer v1,v2,v3;
          Exp     e1,e2;
    case INTconst(v1) then v1;   /* evaluation of an integer node */
                                 /* is the integer value itself */

/* 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 = intAdd(v1,v2);  then v3;

    case SUBop(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); v3 = intSub(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 matchcontinue;

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 `ival` 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 `ival` 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 = inAdd(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 = intAdd(v1,v2)` will be checked. This local equation refers to a pre-defined MetaModelica function called `intAdd` for addition of integer values. For a full set of pre-defined 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 MetaModelica 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`, etc.:

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

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.

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 := matchcontinue inExp
    local Integer v1,v2;
          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 matchcontinue;
 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;
```
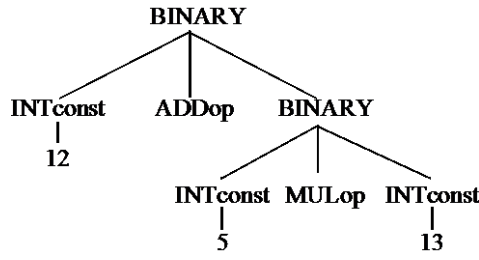
Figure 2-2. A parameterized abstract syntax tree of 12+5*13 in the language Exp1. Compare to the abstract syntax tree in Figure 2-1.

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:=
   matchcontinue 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 matchcontinue;
 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:=
  matchcontinue 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 matchcontinue;
 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:= matchcontinue (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 matchcontinue;
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.

Note: The reader might wonder why we do not directly reference the function input arguments `arg1` and `arg2` in the case, instead of doing a pattern matching to `v1` and `v2`? The reason is a limitation in the current version of the MetaModelica subset compiler which prevents you from accessing function input arguments except in match-expression headers. See also Section 5.14.2.

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:= matchcontinue (op,arg1)
    local Integer v;
    case (NEG(),v) then -v;
  end matchcontinue;
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 treatment of these topics can be found in Chapter 6.

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. 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 not for language specification, but to state a small declarative (i.e., functional) program:

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

The first three lines specifies the name (`fac`) and 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.

2.3.1.1   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 case or in some function called from that case, backtracking takes place, and the next case in the current match-expression is tried instead.

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. 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 6.1.1.

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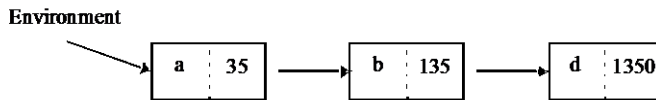
Figure 2-3. An environment represented as a linked list, containing name-value pairs for a, b and d.

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 case 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 is 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:= 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;
```

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)
```

of the first case, and thereby bind `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 corresponding to the two cases. Since the patterns are overlapping and we need to continue trying the next case if the first case fails. We use the **failure**(...) primitive to check for failure of the first equation in the second case.

```
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
        equality(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
        failure(equality(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)::_  ,id)
      equation  equality(id = id2);  then value;
```

It will try to match the `(id2,value) :: _` pattern against the environment argument. The underscore `_` is a "wildcard" pattern that matches anything. 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 local equation `id = id2` is fulfilled, then value will be returned as the result of `lookup`, otherwise the next case will be applied.

The *second* case of `lookup` deals with the case when the identifier might be present in the rest of the list (i.e., not in the leftmost pair). The pattern `(id2,_) :: rest` binds `id2` to the identifier in the leftmost pair, and `rest` to the rest of the list.

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
    failure(equality(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, the following call on the above example environment `env`:

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

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 to update the variable `x` in the above example environment `env`:

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

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)}
```

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

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

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-statement* 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**

which 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;
    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 describe 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
      local Boolean v3;
      equation
        INTval(v1) = eval(env, e1);
        INTval(v2) = eval(env, e2);
        v3 = applyRelop(relop, v1, v2); then BOOLval(v3);
  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 the case 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 defines 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 case 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 ? 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 preceeds 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;
      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
          local Real x,y,z;
        equation
          (env1,v1) = eval(env, e1);
          (env2,v2) = eval(env, e2);
          REAL2(x,y) = type_lub(v1, v2);
          z = applyRealBinop(binop, x, y); 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"
          local Real x,y;
        equation
          (env1,REALval(x)) = eval(env, e);
          y = applyRealUnop(unop, x); then (env1,REALval(y));
      case (env,ASSIGN(id,exp)) /* id := exp;  eval of an assignment node returns
                                    the updated environment and the assigned value.*/
        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.
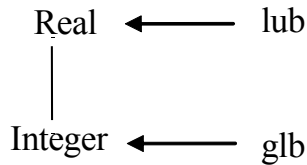
Figure 2-4. Simple type lattice for types integer and real. The least upper bound (lub) is real; the greatest lower bound (glb) is integer.

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 & integer gives real"
  input Value inValue1;
  input Value inValue2;
  output Ty2 outTy2;
algorithm
  outTy2:=
  matchcontinue (inValue1,inValue2)
    local
      Integer x,y;
      Real x2,y2;
    case (INTval(x),INTval(y)) then INT2((x,y));
    case (INTval(x),REALval(y))
        local Real y;
      equation
        x2 = int_real(x); then REAL2((x2,y));
    case (REALval(x),INTval(y))
        local Real x;
      equation
        y2 = int_real(y); then REAL2((x,y2));
    case (REALval(x),REALval(y))
        local Real x,y;  then REAL2((x,y));
  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;
    case (env,id)  // Return value of id in env.
                   // If id not present, add id and return 0
      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()` when referencing the `ASSIGN` constructor from 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 have 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 available at the end of Chapter 6.

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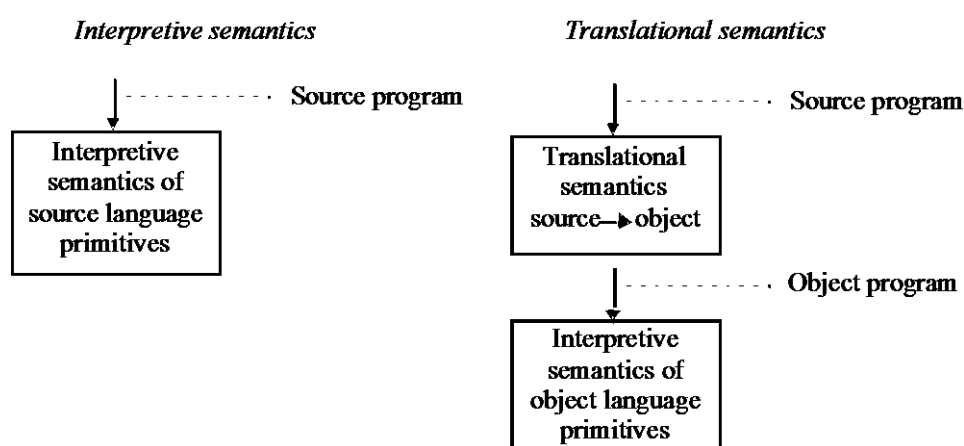
Figure 3-1. A comparison between an interpretive semantics and translational semantics. In an interpretive semantics, the computational meaning of source language primitives are directly defined, e.g. using MetaModelica. In a translational semantics, the meaning is defined as a translation to object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

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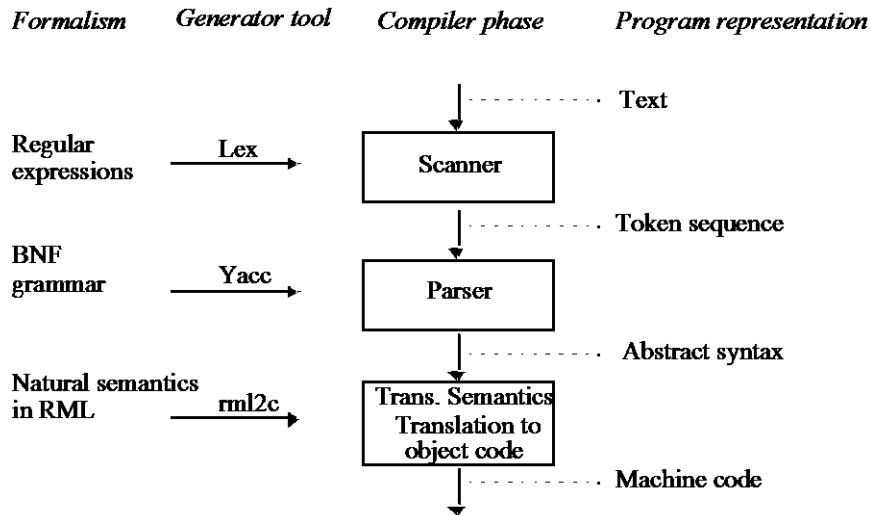
Figure 3-2. Simplified version of compiler generation based on translational semantics. The semantics of a language is specified directly in terms of object code primitives. In comparison to Figure 1-1, the optimization and final code generation phases have been excluded.

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.

```
    case Absyn.BINARY(e1,binop,e2) "Arith binop: simple case, expr2 is just an
                                    identifier or constant:   expr1 binop expr2"
      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.

```
    case Absyn.BINARY(e1,binop,e2) "Arith binop: general case, expr2 is a more
                                    complicated expr:   expr1 binop expr2"
      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
```

which is translated into the 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

    case Absyn.BINARY(e1,binop,e2) // Arith binop: simple case, expr2 is just an
                                   // identifier or constant:  expr1 binop expr2
      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;


    case Absyn.BINARY(e1,binop,e2) // Arith binop: general case, expr2 is a more
                                   // complicated expr:   expr1 binop expr2
      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:=
  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 temporary"
  output Mcode.MOperand outMOperand;
protected
  Integer no;
algorithm
  no = tick();
  outMOperand := Mcode.T(no);
end gentemp;

function listAppend6
  replaceable type Type_a subtypeof Any;
  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:=
  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;
```

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:=
  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;
```

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:

```
    case Absyn.ASSIGN(id,e1)      /* Assignment Statement translation:
                                        map the current state into a new state */
      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 exited 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 following two cases, 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 two 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 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;

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

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

  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;

  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;

  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;

  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;
```

```
      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);

      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);
      case Absyn.SEQ(stmt1,stmt2)               // stmt1 ; stmt2
        equation
          cod1 = transStmt(stmt1);
          cod2 = transStmt(stmt2);
    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;
    case (Mcode.MB(mbinop,mopr))  // Print an MCode instruction
```

```
      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 cases 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;
  case(RCONST(_), _) then RCONST(0.0);        // der of constant
  case(IDENT("time"), _) then RCONST(1.0);    // der of time variable
  case difft(id as IDENT(_), tvars) then      // der of any variable id
    if listMember(id,tvars) then
      CALL(IDENT("der"),list(id))
    else
      RCONST(0.0);
  case (ADD(e1,e2),tvars)                      // (e1+e2)' => e1'+e2'
   equation
    e1prim = difft(e1,tvars);
    e2prim = difft(e2,tvars);  then ADD(e1prim,e2prim);
  case (SUB(e1,e2),tvars)
    equation
     e1prim = difft(e1,tvars);
     e2prim = difft(e2,tvars);
    then SUB(e1prim,e2prim);
  case (MUL(e1,e2),tvars)                      // (e1*e2)' => e1'*e2 + e1*e2'
    equation
     e1prim = difft(e1,tvars);
     e2prim = difft(e2,tvars);
    then PLUS(MUL(e1prim,e2),MUL(e1,e2prim));
  case (DIV(e1,e2),tvars)            // (e1/e2)' => (e1'*e2 - e1*e2')/e2*e2
    equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
    then DIV(SUB(MUL(e1prim,e2),MUL(e1,e2prim)), MUL(e2,e2));
  case (NEG(e1),tvars)                         // (-e1)' => -e1'
    equation
      e1prim = difft(e1,tvars);  then NEG(e1prim);
  case CALL(IDENT("sin"),list(e1)),tvars    // sin(e1)' => cos(e1)*e1'
    equation
      e1prim = difft(e1,tvars);
   then MUL(CALL(IDENT("cos"),list(e1)),e1prim);
  case (AND(e1,e2),tvars)                      // (e1 and e2)' => e1'and e2'
    equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
    then AND(e1prim,e2prim);
 case (OR(e1,e2),tvars)                        // (e1 or e2)' => e1' or e2'
    equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
    then OR(e1prim,e2prim);
  case (LESS(e1,e2),tvars)                     // (e1<e2)' => e1'<e2'
    equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
    then LESS(e1prim,e2prim);
  case (GREATER(e1,e2),tvars)                  // (e1>e2)' => e1'>e2'
    equation
      e1prim = difft(e1,tvars);
      e2prim = difft(e2,tvars);
    then GREATER(e1prim,e2prim);

  // etc...

  end matchcontinue;

end difft;
```

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 semantic cases. The complete PAMTRANS translational semantics is available in Appendix C.3.

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.2, page 194.

Chapter 4

Getting Started – Practical Details

This chapter provides information about a number of technical details that the reader will need to know in order to get started using the MetaModelica mmc generator system. This includes information about where the mmc system resides, how to invoke the `mmc` program generator, how to compile and link generated code, how to run the MetaModelica debugger, etc.

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 will show how to build and run a working calculator that can evaluate constant arithmetic expressions expressed in the Exp1 language. We will also describe how to build an interpreter for a larger language—the PAMDECL language described in Section 2.7.

4.1   Path and Locations of Needed Files

Before one can use the MetaModelica system, some packages need to be installed. The OpenModelica subversion repository is referenced below. In order to use it, you need to provide a username (`anonymous`) and password (`none`, 4 letters).

4.1.1   Windows Dependencies

All dependencies are installed by fetching `https://openmodelica.org/svn/ OpenModelica/installers/Windows/OMDEV` to `C:/OMDev`, and settings 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.

4.1.2   Linux and Mac OSX Dependencies

Use the `rml-mmc` 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 `rml-mmc` from source.

4.1.3   Downloading the Examples

The reader may fetch the example files from `http://openmodelica.org/ metamodelica/exercises/` where the MetaModelica 1.0 related examples from this book are currently located.

4.2   The Exp1 Calculator Again

4.2.1   Running the Exp1 Calculator

Before building the Exp1 calculator it is instructive to show how it can be used. The executable has been named calc, and is invoked by just typing calc at the Unix command prompt (`sen20%10`). Input typed by the user is shown in boldface.

First type `executable` to invoke the calculator, which responds with some trace printout to show that it has initialized and has started parsing text read from the command line.

Then type the expression to be evaluated (here: -5+10-2), followed by pushing the `Enter` key and typing `ctrl-D` (`^D`)[1]. The `ctrl-D` is needed to close the input file (which here is a "terminal"), since the Yacc-generated parser currently expects to read a whole input file before completing the parsing. Finally a trace printout (`[Calc]`) from the evaluator is printed, together with the result (3) of evaluating the expression.

```
>> calc
[Init]
[Parse]
-5+10-2
^D[Eval]
```

Result: 3

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.

```
sen20%11 executable
[Init]
[Parse]
hej+5
Syntax error at or near line 1.
Parsing failed!
```

4.2.2   Building the Exp1 Calculator

Before building the Exp1 calculator, we need to locate the MetaModelica, Lex and Yacc 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).

4.2.2.1   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, a file `main.c` defines the C main program that calls initialization routines, the generated scanner, parser and evaluator, and prints the evaluated result.

---

[1] Windows users may need to use `ctrl-Z` depending on what terminal software is used

4.2.2.2   Generated Source Files

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

- The files `Exp1.c` and `Exp1.h` are generated by the `mmc` translator using the `rml` command. The generated C code that performs evaluation of Exp1 expressions can be found in `Exp1.c`, whereas `Exp1.h` contains tree-building macros to be called by the parser to build abstract syntax trees of input expressions that are passed to the evaluator.

- The files `parser.c` and `parser.h` are generated by Yacc, and contain a parser for Exp1 and token definitions, respectively.

- The file `lexer.c` is generated by Lex, and contains a scanner for Exp1.

4.2.2.3   Library File(s)

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

- The file `rml.h` contains definitions and macros for calling the MetaModelica runtime system and predefined functions (located in `$RMLHOME/include/plain`).

- The file `librml.a` is a library of all MetaModelica runtime system routines and predefined functions (located in `$RMLHOME/lib/plain`).

4.2.2.4   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 `$RMLHOME/{include,lib}`. The usual make dependencies are specified. The command:

```
make −f Makefile.rml executable
```

will build the binary executable of the calculator (called `executable`) whereas the command:

```
make −f Makefile.rml clean
```

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

```
# Makefile.rml
# Makefile for building the Exp1 calculator
CLEAN=Exp1.{c,h,o,srz} main.o
HEADERS=Exp1.h
OBJS=Exp1.o main.o lexer.o parser.o
include ../common.rml
# ../common.rml
ifdef OMDEV
  RMLC=$(OMDEV)/tools/rml/bin/rmlc
  LDLIBS = -lfl -lm -lwsock32
else
  RMLC=rmlc
  LDLIBS = -lm −lpthread
endif

RMLCFLAGS=-g -Wr,-Ono-cps -DRML
COMPILE.rml=$(RMLC) $(RMLCFLAGS) $(CPPFLAGS) -I../rml/ -c
LINK.rml=$(RMLC) -g

.SUFFIXES:      .c .o .rml .h .mo

.c.o:
        $(COMPILE.rml) $<
.rml.c:
        $(COMPILE.rml) +C $<
.rml.h:
```

```
        $(COMPILE.rml) +C $<
.rml.o:
        $(COMPILE.rml) $<

.mo.c:
        $(COMPILE.rml) +C $<
.mo.h:
        $(COMPILE.rml) +C $<
.mo.o:
        $(COMPILE.rml) $<

all: clean executable

# LEXER

lexer.o:  lexer.c parser.h $(HEADERS)
lexer.c:  lexer.l
        flex -t -l lexer.l >lexer.c

# PARSER

parser.o:  parser.c parser.h $(HEADERS)
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 executable parser.c lexer.c parser.h
 lexer.h rtest yacclib.c *~

executable: $(HEADERS) $(OBJS)
        $(LINK.rml) $(OBJS) $(LDLIBS) -o executable

run: executable
        cat program.txt
        ./executable 10 < program.txt
```

4.2.3   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 file `main.c`.

4.2.3.1   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` in `Exp1.h` builds abstract syntax integer leaf nodes and is generated by mmc when processing the abstract syntax definitions in `Exp1.mo`.

  The routine `mk_icon` (from `rml.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 */
%{
#include "parser.h"
#include "rml.h"
#include "Exp1.h"

typedef void *rml_t;
```

```
extern rml_t yylval;

rml_t absyn_integer(char *s);

%}

digit           ("0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9")
digits          {digit}+
junk            .|\n

%%

{digits}        { yylval=absyn_integer(yytext); return T_INTCONST;}
"+"             return T_ADD;
"-"             return T_SUB;
"*"             return T_MUL;
"/"             return T_DIV;
"("             return T_LPAREN;
")"             return T_RPAREN;
{junk}+             ;

%%

omc_t absyn_integer(char *s)
{
  return (rml_t) Exp1__INTconst(mk_icon(atoi(s)));
}
```

4.2.3.2   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` are generated by mmc from the definition of the Exp1 abstract syntax in the package `Exp1` that can be found in the file `Exp1.mo`. The definition of these can be found in `Exp1.h`. Leaf nodes such as `INTconst` are returned by the scanner.

```
/* file  parser.y */
%{
#include <stdio.h>
#include "rml.h"
#include "Exp1.h"

#define YYSTYPE rml_t
extern rml_t absyntree;

%}

%token T_INTCONST

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

%%

/* 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);}

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

4.2.3.3   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 x1;      end INTconst;
  record  ADDop  Exp x1;  Exp x2;   end ADDop;
  record  SUBop  Exp x1;  Exp x2;   end SUBop;
  record  MULop  Exp x1;  Exp x2;   end MULop;
  record  DIVop  Exp x1;  Exp x2;   end DIVop;
  record  NEGop  Exp x1;            end NEGop;
end Exp;



 /* Evaluation semantics  of Exp1 */


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

end eval;
```

4.2.3.4   main.c

See Section 4.2.4 for more information.

4.2.4   Calling MetaModelica from C — main.c

The main program in a MetaModelica-based application can be written either in C or in MetaModelica itself. Here we present an example where the main program is in C.

The main program ties the different packages together and initializes the MetaModelica runtime system. It may also take care of possible command line arguments if the generated application needs those.

In this particular program, the procedure `Exp1_5finit` is first called to in order to initialize the MetaModelica runtime system. In fact, for each package `M` written in this implementation of MetaModelica, the C main program must call `M_5finit();` for initialization. Then the printouts [`Init`] and [`Parse`] are produced, after which the user is expected to type in an expression, which is parsed and scanned by `yyparse`. The abstract syntax tree is built by the parser and placed into the global variable `absyntree`.

The parameter passing facilities between C code and MetaModelica functions are tied to the `mmc` implementation and a bit primitive. The abstract syntax tree need to be passed to the Modelica function `Exp1.eval` for evaluation, which is the main functionality in our calculator. To do this, the tree is placed into the global location `rml_state_ARGS[0]` which transfers the first argument to `Exp1.eval` through the call `rml_prim_once(RML_LABPTR(Exp1__eval))` which returns a non-zero value if the evaluation is successful. The integer result of the evaluation is placed in the global variable `rml_state_ARGS[0]`. Note that the result must be converted from the MetaModelica tagged integer representation to the ordinary C integer representation before being printed. This conversion is handled by `RML_UNTAGFIXNUM`.

The special MetaModelica runtime system procedures and locations referred to, such as `rml_prim_once`, `rml_state_ARGS`, `RML_LABPTR`, etc., are all declared in the include file `omc.h`. The file `main.c` follows below.

```
/* file main.c */
/* Main program for the small Exp1 evaluator */

#include <stdio.h>
#include <rml.h>
#include "Exp1.h"

rml_t absyntree;

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

main()
{
  int res;
```

```
  /* Initialize the Modelica packages */

  printf("[Init]\n");
  exp1_5finit();

  /* Parse the input into an abstract syntax tree (in Modelica form)
     using yacc and lex */

  printf("[Parse]\n");
  if (yyparse() !=0)
  {
    fprintf(stderr,"Parsing failed!\n");
    exit(1);
  }

  /* Evalute it using the Modelica relation "eval" */

  printf("[Eval]\n");
  rml_state_ARGS[0]= absyntree;
  if (!rml_prim_once(RML_LABPTR(Exp1__eval)) )
  {
    fprintf(stderr,"Evaluation failed!\n");
    exit(2);
  }

  /* Present result */

  res=RML_UNTAGFIXNUM(rml_state_ARGS[0]);
  printf("Result: %d\n", res);
}
```

4.2.5   Generated Files and Library Files

We have already mentioned the five generated files `lexer.c`, `parser.h`, `parser.c`, `Exp1.h`, and `Exp1.c` in Section 4.2.2.2. The MetaModelica system generates `Exp1.h` and `Exp1.c`. Here we will present the header file `Exp1.h` in more detail. The file `Exp1.c` contains optimized C implementations of the `Exp1` MetaModelica functions, which is rather unreadable C code that is not so interesting to look at.

  Additionally, we describe the parts of the header file `rml.h` that are used for building and printing abstract syntax trees.

4.2.5.1   Exp1.h

The header file `Exp1.h` contains declarations that makes it possible to call entities declared in the interface section of the `Exp1` Modelica package. These include the `Exp1.eval` function referred to through the label `Exp1__eval`, and abstract syntax tree constructors `Exp1.NEGop`, `Exp1.DIVop`, etc. which can be called through the macros `Exp1__NEGop`, `Exp1__DIVop`, etc. respectively.

```
/* interface Exp1 */
extern void Exp1_5finit();
extern RML_FORWARD_LABEL(Exp1__eval);
#define Exp1__NEGop_3dBOX1 5
#define Exp1__NEGop(X1) (mk_box1(5,(X1)))
#define Exp1__DIVop_3dBOX2 4
#define Exp1__DIVop(X1,X2) (mk_box2(4,(X1),(X2)))
#define Exp1__MULop_3dBOX2 3
#define Exp1__MULop(X1,X2) (mk_box2(3,(X1),(X2)))
#define Exp1__SUBop_3dBOX2 2
#define Exp1__SUBop(X1,X2) (mk_box2(2,(X1),(X2)))
#define Exp1__ADDop_3dBOX2 1
#define Exp1__ADDop(X1,X2) (mk_box2(1,(X1),(X2)))
#define Exp1__INTconst_3dBOX1 0
#define Exp1__INTconst(X1) (mk_box1(0,(X1)))
```

4.2.5.2   rml.h

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

The routines `mk_icon`, `mk_rcon`, `mk_scon` create MetaModelica representations for integers, real numbers and strings, respectively, whereas `print_icon`, `print_rcon`, and `print_scon` can print Modelica integers, real numbers and strings.

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

Finally, the routines `mk_box0` to `mk_box9` construct abstract syntax tree nodes of arity 0 to 9. These should not be called directly, however. Instead use the abstract syntax building routines, one for each node type, which are declared in the file `Exp1.h`.

4.3   An Evaluator for PAMDECL

4.3.1   Running the PAMDECL Evaluator

The executable is named `executable`, and is invoked by typing `executable` at the Unix prompt (`sen20%10`). Input typed by the user is shown in boldface.

```
sen20%10 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:

```
sen20%11 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
```

4.3.2   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`, `ScanParseRML.c` and `Makefile.rml`.

The file contents are listed in Appendix C.2 and can be also be obtained by following the instructions in Section 4.1.3.

The executable is built by typing:

```
sen20%12 make -f Makefile.rml executable
```

4.3.3   Calling C from MetaModelica

The file `ScanParse.mo` looks somewhat weird. It does not contain the usual package implementation section. In the makefile one also notices that it is not compiled using `mmc`. Instead we supply the body for `ScanParse.mo` through the file `ScanParseRML.c`, which in turn is compiled in a regular way. This is the trick to use when there is a need to call C from MetaModelica.

This is how you do it in PAMDECL:

- In `ScanParse.mo` specify the functions (C functions) that are to be implemented in C. In this case it is a function (function) that takes no arguments and returns an `Absyn.prog`.
- In `ScanParseRML.c` we need to implement the functions (function) specified in `ScanParse.mo`. This is done by typing the code for the function between `RML_BEGIN_LABEL(ScanpParse__relationname)` and `RML_END_LABEL`.
- One also needs to add the constructor `ScanParse_5finit(void)` for `ScanParse.mo`, which in this case does nothing.

If one wants the function to fail call `RML_TAILCALLK(omcFC)` or call `RML_TAILCALLK(omcSC)` if one want it to succeed.

Values are returned through the variable `rmlA0`. Values submitted to the function (function) can be retrieved from `rmlA0` through `rmlA9`. Before the values can be retrieved or returned they have to be untagged or tagged, e.g. get a string parameter.

```
char *first_param = RML_STRINGDATA(rmlA0);
```

or return a string constant

```
rmlA0 = (void *) mk_scon("Hello, world!");
```

Chapter 5

Comprehensive Overview of the Current MetaModelica Subset

This chapter gives a comprehensive overview of all MetaModelica constructs, from basic primitives such as as characters and lexical units including identifiers, literals, and operators, to expressions, functions, and declarations.

5.1   MetaModelica Constructs to be Depreciated

The current MetaModelica 1.0 subset contains several constructs which will eventually eventually be removed from the MetaModelica language. They are needed before compiler support for better alternatives has been implemented. The constructs to be depreciated are the following:

- matchcontinue-expressions will probably eventually be replaced by match-expressions and exception handling or match-expressions with guards.
- Real number arithmetic operators containing a dot (+., -., *., /., etc.) will be replaced by ordinary overloaded arithmetic operators (+, - ,* , /, etc.)
- The String ==& equality operation and +& string append operator will be replaced by the overloaded operators == and + respectively.
- The equality(...) operator will be removed.
- Some additional smaller items.

5.2   Character Set

The character set of the Modelica language is not yet completely specified. However, in practice the currently available Modelica tools work well for code written in the 8-bit Latin-1 character set. Most of the first 128 characters of Latin-1 are equivalent to the 7-bit ASCII character set.

5.3   Comments

There are three kinds of comments in Modelica which are not lexical units in the language and therefore are ignored by a Modelica translator. The comment syntax is identical to that of Java. The following comment variants are available:

`// comment`	Characters from `//` to the end of the line are ignored.
`/* comment */`	Characters between `/*` and `*/` are ignored, including line terminators.
`/** comment */`	Characters between `/**` and `*/` are ignored, including line terminators. These are documentation comments that come immediately before declarations and can be included in automatically generated documentation. However, currently available Modelica tools primarily support another mechanism for documentation, so-called documentation strings described below, which can be attached after each declaration.

Modelica comments do not nest, i.e., /* */ cannot be embedded within /* */. The following is *invalid*:

```
/* Commented out – erroneous comment, invalid nesting of comments!
   /* This is a interesting model */
   function interesting
   ...
   end interesting;
*/
```

There is also a kind of "documentation comment," really a *documentation string*, that is part of the Modelica language and therefore not ignored by the Modelica translator. Such "comments" may occur at the ends of declarations, at the beginnings of function definitions, or immediately after any equation. For example:

```
function foo  "This is a function comment"
   ...
   Real x  "the variable x is used for ...";
   ...
end foo;
```

5.4   Identifiers, Names, and Keywords

Identifiers are sequences of letters, digits, and other characters such as underscore, which are used for *naming* various items in the language. Certain combinations of letters are *keywords* represented as *reserved* words in the Modelica grammar and are therefore not available as identifiers.

5.4.1   Identifiers

Modelica *identifiers*, used for naming classes, variables, constants, and other items, are of two forms. The first form always start with a letter or underscore (_), followed by any number of letters, digits, or underscores. Case is significant, i.e., the names Inductor and inductor are different. The following BNF-like rules define MetaModelica identifiers, where curly brackets {} indicate repetition zero or more times, and vertical bar | indicates alternatives.

```
IDENT    = NONDIGIT { DIGIT | NONDIGIT } |  Q-IDENT
NONDIGIT = "_" | letters "a" to "z" | letters "A" to "Z"
DIGIT    = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
S-ESCAPE = "\'" | "\"" | "\?" | "\\" |
           "\a" | "\b" | "\f" | "\n" | "\r" | "\t" | "\v"
```

5.4.2   Names

A *name* is an identifier with a certain interpretation or meaning. For example, a name may denote an Integer variable, a Real variable, a function, a type, etc. A name may have different meanings in different parts of the code, i.e., different scopes.

5.4.3   MetaModelica Keywords

The following MetaModelica *keywords* are reserved words and may not be used as identifiers:

_		as	and	annotation	algorithm
block		case	constant		
else					end
equality	equation			external	false
failure	fail			function	if

input	list	local	match	matchcontinue
not	or		output	package
		protected	public	record
		subtypeof	then	true
tuple	type	uniontype		

5.5   Predefined Types

The predefined built-in based types of MetaModelica 1.0 are `Real`, `Integer`, `Boolean`, and `String`. The machine representations of the *values* of these predefined types have the following properties:

`Real`	IEC 60559:1989 (ANSI/IEEE 754-1985) double format, at least 64-bit precision.
`Integer`	typically two's-complement 32-bit integer. (But currently 31 bit integer in the boxed Integer type.)
`Boolean`	`true` or `false`.
`String`	string of 8-bit characters.
`list<eltype>`	list of element type

Note that for argument passing of values when calling external functions in C from MetaModelica, `Real` corresponds to `double` and `Integer` corresponds to `int`.

5.5.1   Literal Constants

Literal constants are unnamed constants that have different forms depending on their type. Each of the predefined types in MetaModelica has a way of expressing unnamed constants of the corresponding type, which is presented in the ensuing subsections. Additionally, array literals and record literals can be expressed.

5.5.2   Floating Point Numbers

A floating point number is expressed as a decimal number in the form of an optional sign (+ or −), a sequence of decimal digits optionally followed by a decimal point, optionally followed by an exponent. At least one digit must be present. The exponent is indicated by an E or e, followed by an optional sign (+ or −) and one or more decimal digits. The range is that of IEEE double precision floating point numbers, for which the largest representable positive number is 1.7976931348623157E+308 and the smallest positive number is 2.2250738585072014E−308. For example, the following are floating point number literal constants:

```
22.5,  3.141592653589793, 1.2E-35, -56.08
```

The same floating point number can be represented by different literals. For example, all of the following literals denote the same number:

```
13.,  13E0,  1.3e1,  .13E2
```

5.5.3   Integers

Literals of type `Integer` are sequences of decimal digits, e.g. as in the integer numbers 33, 0, 100, 30030044, or negative numbers such as −998. The range depends on the C compiler implementation of integers (Modelica compiles to C), but typically is from −2,147,483,648 to +2,147,483,647 for a two's-complement 32-bit integer implementation. Currently only 31-bit integers are supported as the boxed Integer type in MetaModelica.

5.5.4   Booleans

The two `Boolean` literal values are `true` and `false`.

5.5.5   Strings

String literals appear between double quotes as in `"between"`. Any character in the MetaModelica language character set apart from double quote (`"`) and backslash (`\`), but including nonprintable characters like new-line, backspace, null, etc., can be *directly* included in a string without using an escape code. Certain characters in string literals are represented using escape codes, i.e., the character is preceded by a backslash (`\`) within the string. Those characters are:

`\'`	single quote—may also appear without backslash in string constants.
`\"`	double quote
`\?`	question-mark—may also appear without backslash in string constants.
`\\`	backslash itself
`\a`	alert (bell, code 7, ctrl-G)
`\b`	backspace (code 8, ctrl-H)
`\f`	form feed (code 12, ctrl-L)
`\n`	new-line (code 10, ctrl-J)
`\r`	return (code 13, ctrl-M)
`\t`	horizontal tab (code 9, ctrl-I)
`\v`	vertical tab (code 11, ctrl-K)

For example, a string literal containing a tab, the words: `This is`, double quote, space, the word: `between`, double quote, space, the word: `us`, and new-line, would appear as follows:

```
"\tThis is\" between\" us\n"
```

Concatenation of string literals in certain situations (see the Modelica grammar) is denoted by the + operator in Modelica, e.g. `"a"` + `"b"` becomes `"ab"`. This is useful for expressing long string literals that need to be written on several lines.

5.5.7   Array Literals in MetaModelica

Array literals can be created using the constructor { }:

```
{1,2,3},     {3.14, 58E-6}
```

It is also possible to use the `fill` function to create an array, e.g.:

```
fill(0,35)        // Create an Integer array of 35 elements filled with 0
```

Arrays of other types can be created by specifying another fill value:

```
fill(3.14, 99)   // Create an array of 99 Real elements of value 3.14
```

5.5.8   List Literals

List literals can be expressed using the list constructor `list(...)` or by the array constructor { } which creates an array that is 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 construct an array that is automatically converted to a list depending on the type context, e.g. if it is past to a function formal parameter of list type, or assigned to a variable of list type.

```
{1,2,3},     {3.14, 58E-6}
```

5.5.9   Record Literals

Record literals can be expressed using the record constructor functions automatically defined as a consequence of record declarations (inside union types). Below is an example record literal of a complex number based on the record `Complex`:

```
Complex(1.3, 4.56)
```

5.6   Operator Precedence and Associativity

Operator precedence determines the order of evaluation of operators in an expression. An operator with higher precedence is evaluated before an operator with lower precedence in the same expression. For example, relational operators have higher precedence than logical operators, e.g.:

```
Xwithin := x>35.3 and x<=999.6;
```

Assuming x has the value 55.0, then both relational terms are first evaluated to `true`, eventually giving the value `true` to be assigned to the variable `Xwithin`. The multiplication operator * has higher precedence than the subtraction operator, causing the following expression to have the value 45, not zero:

```
10 * 5 – 5
```

Parentheses can be used to override precedence, e.g. causing the expression below to evaluate to zero:

```
10 * (5 – 5)
```

The associativity determines what happens when operators with the same precedence appear next to each other. Left-associative operators evaluate the leftmost part first, e.g. the expression:

```
x + y + w
```

is equivalent to

```
(x + y) + w
```

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 5-1. Operators.

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}
real power of	^.	x ^. 2
integer multiplicative	* /	2*3 2/3
real multiplicative	*. /.	2.1 *. 3.2
integer additive	+ – +*expr* -*expr*	a+b, a-b, +a, -a
real additive	+. -. +. *expr* -. *expr*	a+.b, a-.b, +.a, -.a
integer relational	< <= > >= == <>	a<b, a<=b, a>b, ...
real relational	<. <=. >. >=. ==. <>.	a<.b, a<=.b, a>.b, ...
string equality	==&	str1 ==& str2
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, `equality(expr1 = expr2)`, giving fail or succeed, which can be applied to values of primitive data types as well as to values of structured types such as arrays, lists, and trees.

The above operators correspond to and can be called using the following function names, which are mentioned below together with a few additional builtin functions:

The following are built-in common mathematical functions:

Table 5-2. Built-in common mathematical functions.

`sin(`*u*`)`	sine
`cos(`*u*`)`	cosine
`tan(`*u*`)`	tangent (*u* shall not be: $..., -\pi/2, \pi/2, 3\pi/2,...$)
`asin(`*u*`)`	inverse sine $(-1 \leq u \leq 1)$
`acos(`*u*`)`	inverse cosine $(-1 \leq u \leq 1)$
`atan(`*u*`)`	inverse tangent
`atan2(`*u1,u2*`)`	four quadrant inverse tangent
`sinh(`*u*`)`	hyperbolic sine
`cosh(`*u*`)`	hyperbolic cosine
`tanh(`*u*`)`	hyperbolic tangent
`exp(`*u*`)`	exponential, base *e*
`log(`*u*`)`	natural (base *e*) logarithm $(u > 0)$
`log10(`*u*`)`	base 10 logarithm $(u > 0)$

Boolean operations:

```
boolAnd, boolOr, boolNot
```

Integer operations:

```
intAdd, intSub, intMul, intDiv

intMod, intAbs, intNeg, intMax, intMin

intLt, intLe, intEq, intNe, intGe, intGt, int_real, intString
```

Real number operations:

```
realAdd, realSub, realMul, realDiv

realMod, realAbs, realNeg, realMax, realMin

realLt, realLe, realEq, realNe, realGe, realGt, realInt, realString

realCos, realSin, realAtan, realExp, realLn, realFloor, realInt, realPow
```

String operations:

```
stringLength, stringGet, stringAppend

stringInt, stringList, listString
```

For a complete set of builtin operators and functions, including their signatures, see Appendix B.

5.7   Arithmetic Operators

MetaModelica supports four binary arithmetic operators in both integer and real variants. The real number operators currently contain a dot.

Table 5-3. Arithmetic operators.

Integer operators		Real operators	
*	Integer multiplication	*.	Real multiplication
/	Integer mivision	/.	Real division
+	Integer addition	+.	Real addition
−	Integer subtraction	−.	Real subtraction

Some of these operators can also be applied to a combination of a scalar type and an array type, which means an operation between the scalar and each element of the array.

Unary versions of the addition and subtraction operators are available, e.g. as in –35 and +84.

5.7.1   Integer Arithmetic

Integer arithmetic in Modelica is the same as in the ISO C language standard, since Modelica is compiled into C. The most common representation of integers is 32-bit two's complement, e.g. see a definition in *C—A Reference Manual*, Section 5.1.1, (Harbison and Steele 1991). This representation is used on widespread modern microprocessors such as Pentium, Sparc, etc., with a minimum representable value of –2,147,483,648 and a maximum value of 2,147,483,647. Note, however, that other representations are also allowed according to the ISO C standard. Note that currently, only 31-bit integer arithemtic is supported by the MetaModelica compiler for boxed integers..

For certain arithmetic operations, regarding both integer and floating point numbers, it can be the case that the true mathematical result of the operation cannot be represented as a value of the expected result type. This condition is called *overflow*, or in some cases *underflow*.

In general, neither the MetaModelica language nor the C language specify the consequences of overflow of an arithmetic operation. One possibility is that an incorrect value (of the correct type) is produced. Another possibility is that program execution is terminated. A third possibility is that some kind of exception or trap is generated that could be detected by the program in some implementation-dependent way.

For the common case of two's complement representation, integer arithmetic is modular—meaning that integer operations are performed using a two's-complement integer representation, but if the result exceeds the range of the type it is reduced modulo the range. Thus, such integer arithmetic never overflows or underflows but only wraps around.

Integer division, i.e., division of two integer values, truncates toward zero with any fractional part discarded (e.g. div(5,2) becomes 2, div(-5,2) becomes -2). This is the same as in the C language according to the C99 standard. According to the earlier C89 standard, integer division for negative numbers was implementation dependent.

Division by zero in Modelica causes unpredictable effects, i.e., the behavior is undefined. In typical cases execution is aborted. Division by zero or other similar faults in MetaModelica generates a failure which can be handled within a match-expression, see Section 5.14.6.

5.7.2   Floating Point Arithmetic

Analogous to the case for integer arithmetic, floating point arithmetic in Modelica is specified as floating point arithmetic in the ISO C language. Values of the Modelica Real type are represented as values of the double type in ISO C, and floating point operations in Modelica are compiled into corresponding doubleprecision floating point operations in C. Even if not strictly required by the ISO C standard, most C implementations have adopted the IEEE standard for binary floating point arithmetic (ISO/IEEE Std

754-1985), which completely dominates the scene regarding C implementations as well as floating point instructions provided by modern microprocessors. Thus, we can for practical purposes assume that Modelica follows ISO/IEEE Std 754-1985. Real values are then represented as 64-bit IEEE floating point numbers. The largest representable positive number in the IEEE double precision representation is 1.7976931348623157E+308 whereas the smallest positive number is 2.2250738585072014E−308.

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.

5.8   Equality, Relational, and Logical Operators

MetaModelica supports the standard set of relational and logical operators, all of which produce the standard boolean values `true` or `false`.

Table 5-4. Integer and real relational operators.

Integer Relational Operators		Real Relational Operators	
>	greater than	>.	greater than
>=	greater than or equal	>=.	greater than or equal
<	less than	<.	less than
<=	less than or equal to	<=.	less than or equal to
==	equality within expressions	==.	equality within expressions
<>	Inequality	<>.	Inequality

The equality and relational operators apply only to scalar arguments. Relational operators are typically used within if-expressions, or to compute the value of a `Boolean` variable, e.g.:

```
x = if v1<v2 then ...  ;
boolvar2 := v3 >= v35;
```

A single equals sign = is never used in relational expressions, only in equations and in function calls using named parameter passing.

Table 5-5. Equality sign in equations and named parameter passing.

=	equality within equations
=	equality of named parameters to arguments at function calls using named parameter passing

Comparison of strings or structured values can be done using an equation of the form `equality(a=b)` which succeeds or fails. See numerous examples of comparision of identifiers in lookup functions described in this document.

The following logical operators are defined:

Table 5-6. Logial operators.

not	negation, unary operator
and	logical and
or	logical or

Standard Modelica is free to use *any order* in evaluation of expression parts as long as the evaluation rules for the operators in the expression are fulfilled.

Concerning the logical operators `and`, `or` in boolean expressions, one possibility is short-circuit evaluation, i.e., the expression is evaluated from left to right and the evaluation is stopped if evaluation of further arguments is not necessary to determine the result of the boolean expression. Thus, if the

variable `b1` in the expression below has the value `true`, then evaluation of `b2` and `b3` would not be necessary since the result will be `true` independent of their values. On the other hand, we cannot rely on this order—evaluation might start with `b3` and involve all three variables. However, this does not really matter for the user since Modelica is a declarative language, and the result of evaluation is the same in all these cases..

```
boolvar   :=  true and false;
boolvar2  :=  not boolvar;
boolvar3  :=  b1 or b2 or b3;
```

5.8.2 String Concatenation in MetaModelica

The `stringAppend` operation is a built-in string concatenation function in MetaModelica, both for string variables and literal string constants.

```
String val1 = "This is";
String val2 = " a ";
String concatvalue = stringAppend(stringAppend(val1, val2), stringAppend("rather
"," long string"));
// The value becomes: "This is a rather long string"
```

5.8.3 The Conditional Operator—if-expressions

The conditional operator in Modelica provides a single expression that computes one out of two expressions dependent on the value of the condition. The general syntactic form is shown below:

if *condition* **then** *expression1* **else** *expression2*

Both the `then`-part and the `else`-part of the conditional expression must be present. Conditional expressions can be nested, i.e., *expression2* can itself be an if-expression.

A conditional expression is evaluated as follows:

- First the *condition* is evaluated, which must be a boolean expression. If *condition* is true, then *expression1* is evaluated and becomes the value of the if-expression. Otherwise *expression2* is evaluated and becomes the value of the if-expression.
- The result expressions, i.e., *expression1* and *expression2*, must have assignment-compatible types. This means that the type of one result expression must be assignable to the type of the other result expression, which defines the type of the conditional expression.

The following equation contains a conditional expression with a conditional operator on the right-hand side:

```
value = (if a+b<5 then firstvalue else secondvalue);
```

5.9 Built-in Special Operators and Functions

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. The following operators are supported:

Table 5-7. Special operators with function syntax.

`failure(...)`	If the argument fails this function succeeds.
`equality(a = b)`	Same as a general structural equality equation a = b. Either fails or succeeds.

`list(...)` or `{...}`	List constructor. Also use curly braces { }
`fill(value, noElements)`	Array constructor that creates a vector of length noElements filled with value.

5.10 Order of Evaluation

Evaluation order is currently left-to-right, but will become unspecified in the future when the MetaModelica compiler is upgraded to also support full Modelica.

5.11 Expression Type and Conversions

All expressions have a *type*. The expression type is obtained from the types of its constituent parts, e.g. variables, constants, operators, and function calls in an expression.

5.11.1 Type Conversions

Meta-Modelica is a strongly typed language. This means that type compatibility is checked at compile time in almost all cases, and at run-time in the remaining cases. MetaModelica prevents incompatible left-hand and right-hand sides in equations as well as incompatible assignments by not allowing anything questionable.

The language also provides a few checking and *type conversion* operations for cases when the compatibility of a type can be determined only at run-time, e.g. to check the size of a variable-length array, or when we want to explicitly convert a type, for example, when assigning a `Real` value to an `Integer` variable. We discuss these conversions in terms of assignment, sometimes called *assignment conversion*, but what is said here is also applicable to conversions between left-hand sides and right-hand sides of equations, and conversions when passing actual arguments to formal parameters at function calls.

5.11.1.1 Implicit Type Conversions in Modelica

Sometimes a type can be converted without any explicit action from the Modelica programmer. The only case in standard Modelica when this happens is *implicit conversion* of integer operands when used together with floating point operands in an expression. However, in the current MetaModelica, all type conversions must by explicit, but implicit conversions are planned for the next major version of MetaModelica.

5.11.1.2 Implicit Type Conversions in MetaModelica

In the current MetaModelica, implicit type conversions are not available – all type conversions are explicit. For example, use `intReal(i)` to convert from Integer to Real or `realInt(r)` from Real to Integer.

5.11.1.3 Explicit Type Conversions

Explicit type conversions are needed when implicit conversions are not enough or are not available, for example, when converting from a `Real` to an `Integer`.

5.12   Global Constant Variables

Global constants can be declared in MetaModelica through the `constant` keyword, e.g. as below where the `initEnv` variable is set to the empty list:

```
constant initEnv = {};
```

5.13   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 tree structures. First we will take a look at the primitive data types.

5.13.1   Primitive Data Types

The MetaModelica language provides a basic set of primitive types found in most programming languages:

- `Boolean`—booleans, e.g. `true`/`false`.
- `Integer`—integers, e.g. `-123`. (Normally 32-bit integers in Modelica, but 31-bit integers in the current MetaModelica version)
- `Real`—double-precision IEEE floating point numbers, e.g. `3.2E5`.
- `String`—strings of characters, e.g. `"Linköping"`.

5.13.2   Type Name Declarations

Alternate names for types in MetaModelica can be introduced through the `type` declaration, e.g.:

```
type  Identifier   = String;
type  IntConstant  = Integer;
type  MyValue      = Real;
```

5.13.3   Tuples

Tuples 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.
- `(55,"Hello",INTconst(77))` — a 3-tuple of integer, `string`, and `Exp`.

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>;
```

5.13.4   Tagged 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;
```

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 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.

5.13.5   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 kinds of parameterized types:

- Lists – the `list` builtin predefined type constructor, parameterized in terms of the type of the list elements.

- Arrays – parameterized in terms of the array element type.

- Option types – the `option` builtin predefined type constructor, parameterized in terms of the type of the optional value.

Note that all 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 tagged union types, where each "type" in the union type is denoted by a different tag.

5.13.5.1   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 following basic list construction operators are available:

- The *list constructor*: `{el1,el2,el3,...}` and `list(el1,el2,el3,...)` create a list of elements `el1`, `el2`, ... of identical type. Examples:`{}` and `list()` denote the empty list; `{2,3,4}` and `list(2,3,4)` are a list of integers, etc.
- The *empty list* is denoted by `{}`.
- The *list element concatenation* operation `cons(element, lst)` or using 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 139 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 6.1.1.

The types of lists often need to be specified. Named list types can be declared using MetaModelica `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 MetaModelica list type with an unspecified element type `TYpe_elemtype` which is a type parameter (type variable) of the list. Type variable names in MetaModelica are declared as replaceable types being subtypes of the "top" type `Any`.

```
replaceable type Type_elemtype subtypeof Any;
```

```
type ElemList = list<Type_elemtype>;
```

Lists in the MetaModelica language 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 tagged union types, where each type in the union type has a different tag.

5.13.5.2   Arrays

A MetaModelica 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.

Constructing arrays is rather clumsy in MetaModelica. First a list has to be constructed which then is converted to an array, e.g.:

```
vec := listArray({2,4,6,8})
```

Accessing the third element of the array `vec` using the array indexing operation `arrayGet`, where the first element has index 1:

```
arrayGet(vec,3) => 6
```

It is also possible to use the more concise square bracket indexing notation:

```
vec[3] => 6
```

Getting the length of array `vec`:

```
arrayLength(vec) => 4
```

Creating 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
```

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 OneDimBooleanVector = Boolean[:];
```

Multi-dimensional arrays are represented by arrays of arrays, e.g. as in the following declaration of a two-dimensional matrix of real elements.

```
type OneDimRealVector = Real[:];
```

```
type TwoDimRealMatrix = OneDimRealVector[:];
```

Parameterized array types can be expressed using a type parameter declared as a replaceable type, such as `Type_ElemType` in the following example:

```
replaceable type Type_ElemType subtypeof Any;
```

```
type Type_ElemVector = Type_ElemType[:];
```

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 arrayLength  "Compute the length of an array"
  input Type_a[:]  inVec;
  output Integer   outLength;
  replaceable type Type_a subtypeof Any;
end arrayLength;
```

```
function arrayGet   "Extract (indexed access) an array element from the array"
  input Type_a[:]  inVec;
  output Type_a    outElement;
  replaceable type Type_a subtypeof Any;
end arrayGet;
```

The following are signatures of the conversion operations between arrays and lists:

```
function arrayList  "convert from array to list"
  input Type_a[:]      inVec;
  output list<Type_a>  outLst;
  replaceable type Type_a subtypeof Any;
end arrayList;
```

```
function listArray  "Convert from list to array"
  input list<Type_a>  inLst;
  output Type_a[:]     outVec;
  replaceable type Type_a subtypeof Any;
end listArray;
```

5.13.5.3   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
  replaceable type Type_a subtypeof Any;
  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.

5.14   MetaModelica 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 section 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 6.1.2 for a discussion of these functions, also see Appendix B.3.11.

Both positional and named argument-passing at function calls are supported.

5.14.1   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. At most one output formal parameter can be used in a MetaModelica function.

```
function <functionname>
  input  TypeI1 in1;
  input  TypeI2 in2;
  input  TypeI3 in3  "Comment" annotation(...);²
  ...
  output TypeO1 out1;
protected
```

---

² Note: default values and multiple output formal parameters are not yet available in MetaModelica.

<local variables>
```
  ...
algorithm
  ...
    <statements>
  ...
end <functionname>;
```

Comment strings and annotations can be given for any formal parameter declaration, as usual in MetaModelica declarations.

All internal parts of a function are optional; i.e., the following is also a legal function:

```
function <functionname>
end <functionname>;
```

5.14.2 Current Restrictions of MetaModelica Functions

Only two supported forms of functions are supported by the current version of the MetaModelica compiler:

- A function with a body consisting of an assignment statement with output variable(s) on the left hand side and a match-expression on the right hand side.
- A function with a body consisting of simple assignment statements.

An example of the first kind:
```
function evalStmtList "Evaluate a list of statements in an environment.
                       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;
      Absyn.StmtList s,ss;
    case (env,{}) then env;
    case (env, s :: ss)
      equation
        env1 = evalStmt(env, s);
        env2 = evalStmtList(env1, ss); then env2;
  end matchcontinue;
end evalStmtList;
```

An example of the second kind:
```
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;
```

There are also additional restrictions:

- Function formal input and output parameter default values and corresponding assignments are not supported.
- In a function body consisting of a match-expression, formal input parameters may only be referenced *directly* after the match keyword, e.g. `match (inX,inY)...` or `match inZ ...`,

and then only in the order declared in the function header. Formal output parameters may *only* be referenced on the left hand side of the assignment comprising the function body.

5.14.4   Function Calls

Both positional and named argument passing is available both in Modelica and in MetaModelica. Default values of formal parametes are not currently available in MetaModelica. The following examples is used:

```
function polynomialEvaluator
  input  Real A[:];        // Array, size defined at function call time
  input  Real x;
  output Real sum;
  ...
```

5.14.4.1   Positional Argument Passing

Example of positional argument passing:

```
p := polynomialEvaluator({1, 2, 3, 4}, 1.0);
```

5.14.4.2   Named Argument Passing

Example of named argument passing:

```
p := polynomialEvaluator(A={1, 2, 3, 4}, x=21);
```

5.14.5   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.

5.14.6   Generating and Handling Failures/Exceptions

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, 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 rules 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 an expression can be done by the `failure(expr)` call. If `expr` fails, then `failure(expr)` succeeds.

See also Section 5.14.7 below and a discussion of failure versus negation in Section 5.15.2.

5.14.7   Special Properties of matchcontinue

Two important properties of MetaModelica functions are absent for ordinary functions:

- Functions in MetaModelica can fail or succeed.
- Retry is supported between rules in a match-expression.

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 rules 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 2.3.1.1.

This brings us into the topic of rule retry. If there is a failure in a 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.

5.14.8   Argument Passing and Result Values

Any kind of data structure, as well as functions, can be passed as actual arguments in a call to an MetaModelica function. One or more results can be returned from such a call. The issues are discussed in some detail in the following sections.

5.14.8.1   Multiple Arguments and Results

A MetaModelica function may be specified with multiple arguments, multiple results, or both. The syntax is simple, the argument and result formal parameters are just listed, preceded by the `input` and `output` keywords respectively.

5.14.8.2   Tuple Arguments and Results from Relations

We just noted that a MetaModelica function can have multiple arguments and results. This should not be confused with the case where a Modelica tuple type (see Section 5.13.3) 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 :=
  matchcontinue inVal
    local Integer x1,x2;
    case (x1,x2) then (x1+1,x2+1);
  end matchcontinue;
end incrementpair;
```

For example, the call:

```
incrementpair((2,3))
```

gives the result:

```
(3,4)
```

5.14.8.3   Passing Functions as Arguments

Functions can be passed as parameters, 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 /*
 ** 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
  replaceable type Type_a subtypeof Any;
  replaceable type Type_b subtypeof Any;
  function FuncType
    replaceable type Type_b subtypeof Any;
    input Type_a in_a;
    output Type_b out_b;
  end FuncType;
algorithm
  out_bList:=
  matchcontinue (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 matchcontinue;
end listMap;


function main
  ...
  res := listMap({0,1,2}, add1);  /* Pass add1 as a parameter to map */
                                  /* In this example res will be {1,2,3}  */
  ...
end main;
```

5.15   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.

5.15.1.1 Type Variables and Parameterized Types in Relations

We have already presented the notion of parameterized list, array, and option types in Section 5.13.5. Type variables in MetaModelica can only 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 "
 ** 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 out_Type_a;
public
  replaceable type Type_a subtypeof Any;
  replaceable type Type_b subtypeof Any;
algorithm
  out_Type_a :=
  matchcontinue (inTuple)
    local Type_a a;
    case (a,_) then a;
  end matchcontinue;
end tuple2GetField1;
```

5.15.1.2 Local Variables in Match-Expressions in Functions

Variables in MetaModelica functions consisting of match-expressions are normally introduced at the beginning of a match-expression or in math-expression cases and have a scope throughout the case. The only exception 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 type variables which are introduced through replaceable type declarations:

- *Type variables*, which are declared in a replaceable type declaration.

For example, in the function `listThread` below, `Type_a` is a 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
  "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;
public
  replaceable type Type_a subtypeof Any;
algorithm
  outList:=
  matchcontinue (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 matchcontinue;
end listThread;
```

5.15.2   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:

```
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(x))
```

The two operators `not` and `failure()` thus represent different forms of "negation"—negating the boolean value `true`, or negating the failure of a call to a function.

5.16   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.

5.16.1   The Match-Expression Construct

The match-expression construct is closely related to pattern matching constructs 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.

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 :=
    matchcontinue s
      case "one"   then 1;
      case "two"   then 2;
      case "three" then 3;
      case  _      then   0;
    end matchcontinue;
```

Alternatively, an else-branch can be used instead of the last wildcard pattern:

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

or using a match-expression with the match keyword:

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

These are trivial special cases. The general structure and evaluation of match-expressions is described in the following sections.

5.16.1.1   Syntactic Structure of Match-Expressions

The general structure of match-expressions starting with either the matchcontinue or the match keyword is indicated by the template 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> <opt-local-decl>
    equation
      <opt-equations>
    then <expr>;
    ...
  case <pat-expr> <opt-local-decl>
    equation
      <opt-equations>
    then <expr>;
    ...
  else <opt-local-decl>
    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> <opt-local-decl>
    equation
      <opt-equations>
    then <expr>;
   ...
  case <pat-expr> <opt-local-decl>
    equation
      <opt-equations>
    then <expr>;
   ...
  else <opt-local-decl>
    equation
      <opt-equations>
    then <expr>;

end matchcontinue;
```

5.16.1.2   Evaluation of Match-Expressions

Match-expressions 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, that may assign unbound pattern variables (declared in local declarations starting with the `local` keyword) to values during the matching process.

- `matchcontinue`. The match data value is matched against the patterns in the case-branches in the order they are declared. If the matching against a pattern succeeds, the rest of the case-branch is evaluated. If the matching against a pattern fails or the matching succeeds but *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 fails and the else-branch fails (if present), the whole match-expression fails.

- `match`. The match data value 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*.

- 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 (this restriction may be removed in the future by sorting local equations). *Current 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;` (currently not OK). See also Section 5.16.5.

- 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.

- The scope of local variables declared *within* a case-branch extends from the *start* of the same case-branch, i.e. before the `local` keyword, throughout the case-branch. Example: `case (x,y)`

`local Real x,y,z;` Here the scope of the local `x,y` start from the beginning of the case-branch including the `(x,y)`.

5.16.2   Usage Contexts and Allowed Forms of Patterns

Patterns can occur after the `case` keyword, and on the left- and right-hand side of the equality sign in equations, in *matching* or *constructive* contexts, with somewhat different meanings. Patterns obey the following rules:

- Patterns can contain calls to record constructor functions, *not* to other kinds of functions.
- Positional and/or named argument function call syntax can be used in patterns containing constructors, e.g. the positional call `FOO(1,_,2)` is allowed; the named argument call version `FOO(arg1=1,arg3=2)` is also allowed.
- Patterns can contain list constructors {3,5,... }, tuple constructors (...,...,...), etc.
- Patterns can contain literal constants, e.g. `"string2"`, 3.14, 555.
- Patterns can contain the _ wildcard.
- 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)`.
- Only local variables declared in local declarations after the `local` keyword may appear as unbound pattern variables in patterns.

5.16.3   Patterns in Matching Context

The most common usage of patterns is in a *matching context* after the `case` keyword, or at the left hand side of = in a local equation, sometimes on the right-hand side.

For example, regard the pattern `INT(x)` on the left-hand side in the case below:

```
matchcontinue 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)`.

5.16.3.1   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 new 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` sucessfully matches a value, a new variable with the name `newname` (need not be declared) and having a type equivalent to the type of `pattern-expression`, is created and 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;
```

5.16.3.2   Example of Patterns with Positional Matching

The following function `eval`, from Section 2.1.4.2, uses 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 :=
  matchcontinue 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 matchcontinue;

 end eval;
```

5.16.3.3   Example of Named Arguments in Pattern Matching

The MetaModelica 1.0 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.

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 :=
  matchcontinue inExp
    local Integer v1,v2;
          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 matchcontinue;
end eval;
```

5.16.3.4   Patterns in Equations and as Constraints

Patterns in matching context may also occur on left-hand sides of local equations. For example:

```
matchcontinue ...
    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"`.

A pattern in an equation may also be used as a constraint. For example, the variable `x`, which already has a value, is here constrained to having the value `false`, otherwise the equation will fail:

```
false = x;   // x 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.

5.16.4   Patterns in Constructive Context

The pattern examples presented so far have been in a matching context, where an existing data item is matched against a pattern possibly containing unbound pattern variables. Patterns can also be used in a constructive context, where a pattern that contains bound pattern variables indicates the construction of a structured data item. For example, regard the pattern in the case below after the **then** keyword:

```
case  ... then (x, {5,y}, INT(z))
```

If the case matches and succeeds and `x` 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))
```


5.16.5   Forms of Equations in match-expression 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, and operators, but currently not functions:

- `expr = funcName(...)`
- `func_name(...)`
- `var_or_const = expr`
- `equality(expr1 = expr2)`
- `failure(var_or_const = expr)`
- `failure(func_name(...))`
- `failure(expr = funcName(...))`
- `failure(equality(expr1 = expr2))`

The `failure()` operator succeeds if the local equation it operates on fails. The equality operation `equality(expr1 = expr2)` succeeds if the data values are identical. Each of these forms can also be parenthesized.

Chapter 6

Declarative Programming Hints

The focus of this chapter is to present a few special issues and give examples of declarative programming style.

6.1.1   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;
```

6.1.1.1   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 inInteger;
  input Integer in_n;
  input Integer inSumSoFar;
  output Integer outRes;
algorithm
  outRes :=
  matchcontinue (inInteger, in_n, inSumSoFar)
    local Integer i,n,i1,res1;
  case (i,n,_)
    equation
      true = (i>n);   then sumSoFar;
  case (i,n,sumSoFar)
    equation
      false = (i>n);
      i1 = i+1;
      res1 = i+sumSoFar; then sumTail(i1,n,res1);
  end matchcontinue;
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.

A more concise formulation of the above `sumTail` function using if-then-else expressions:

```
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;
```

Another example of a tail-recursive formulation is a revised version of the previous `listThread` function from Section 5.15.1.2, 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
 "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;
public
  replaceable type Type_a subtypeof Any;
algorithm
  outList:=
  matchcontinue (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 matchcontinue;
end listThreadTail;
```

6.1.2   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`, that 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.

6.2   More on the Semantics and Usage of MetaModelica Cases

Below we present a number of issues regarding the semantics and usage of MetaModelica match-expression cases.

6.2.1   Logically Overlapping Match Cases

A programming language specification in MetaModelica are often written in such a way that the local equations of different cases in a function are logically overlapping. For example, the predicates x<5 and 3 x<10 are logically overlapping since there are values of x, in the interval [3,5) that satisfy both predicates.

Below we specify a function `func`, which is specified to return x+10 when x<5, and x+20 for $3 \leq$ x<10. This is logically ambiguous in the interval $3 \leq$ x < 5 where both alternatives are valid.

```
function func
  input Real in_x;
  output Real out_y;
algorithm
  out_y :=
  matchcontinue in_x
     local Real x,y;
    case x                // x < 5
      equation
        true = x<5; then x+10;
    case x
      equation            // x>=3 and x<10
        true = (x>=3);
        true = (x<10);  then x+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 $3 \leq$ x < 5. Thus, the first case giving the value x+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.

However, from a strictly logical point of view, from classical Natural Semantics style, ambiguous cases in specifications are inconsistent and should be avoided..

Anyway, the style of specification with more specific conditions first and more general cases later makes sense from a logical point of view when interpreted together with MetaModelica's top-down left-to-right search rule— but is regarded as logically incorrect by purists because of the overlap. It also has the disadvantage that local referential transparency is destroyed, i.e., the semantics of the function is changed if the ordering of the cases is changed. Such a set of cases can be converted to a semantically equivalent set of clumsier non-overlapping cases. Negated conjunctions must then be added to overlapping cases.

6.2.2   Using a Default Match Case in Match-Expressions

There is 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 :=
    matchcontinue inNode
      case UNFOLD(_) then true;
      case ARITH(_)  then false;
      case PTR(_)    then false;
      case ARR(_,_)  then false;
      case REC(_)    then false;
    end matchcontinue;
end function;
```

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 case in match-expressions ensures that the special cases will match if they occur—before the default case which always matches. The logical specification purist will unfortunately regard such a specification as logically incorrect because of the overlap. MetaModelica solves this problem by the use of a general matching pattern _ , as in the example below:

```
function isunfold
  input  Ty      inNode;
  output Boolean outRes;
algorithm
  outRes :=
    matchcontinue inNode
      case UNFOLD(_) then true;
      case _         then false;
    end matchcontinue;
end function;
```

6.3   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, and a function that can be used as a conditional expression (if_) construct within a single MetaModelica case. The functions are the following:

- if_
- listReduce
- listMap
- listFold

6.3.1   If-Expressions Using if_

The `if_` function makes it possible in many cases to avoid having the then-part and the else-part as separate cases. However, the drawback is the both the then-part and the else-part are always evaluated, since they are passed as arguments to the `if_` function.

The function takes a boolean and two values. Returns the first value (second argument) if the Boolean value is true, otherwise the second value (third argument) is returned.

```
if_(true,"a","b") => "a"

function if_
  input Boolean inBoolean1;
  input Type_a inType_a2;
  input Type_a inType_a3;
  output Type_a outType_a;
public
  replaceable type Type_a subtypeof Any;
algorithm
  outType_a:=
  matchcontinue (inBoolean1,inType_a2,inType_a3)
    local Type_a r;
    case (true,r,_) then r;
    case (false,_,r) then r;
  end matchcontinue;
end if_;
```

6.3.2   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
  input list<Type_a> inType_alist;
  input FuncType inFunc;
  output Type_a outType_a;
public
  replaceable type Type_a subtypeof Any;
  function FuncTyp
    input Type_a inType_a1;
    input Type_a inType_a2;
    output Type_a outType_a;
  end FuncType;
algorithm
  outType_a:=
  matchcontinue (inType_alist,inFunc)
    local
      Type_a e,res,a,b,res1,res2;
      FuncType r;
      list<Type_a> xs;
    case (list(e),r) then e;
    case (list(a,b),r)
      equation
        res = r(a, b); then res;
    case (a :: b :: (xs = _ :: _),r)
      equation
        res1 = r(a, b);
        res2 = listReduce(xs, r);
        res = r(res1, res2); then res;
  end matchcontinue;
end listReduce;
```

6.3.3   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
  input list<Type_a> inVtype_alist;
  input FuncType inFunc;
  output list<Type_b> out_vtype_blist;
public
  replaceable type Type_a subtypeof Any;
  replaceable type Type_b subtypeof Any;
  function FuncType
    input Type_a inType_a;
    output Type_b outType_b;
  end FuncType;
algorithm
  out_vtype_blist:=
  matchcontinue (inVtype_alist,inFunc)
    local
      Type_b f_1;
      list<Type_b> r_1;
      Type_a f;
      list<Type_a> r;
      FuncType fn;
    case ({},_) then {};
    case (f :: r,fn)
      equation
        f_1 = fn(f);
        r_1 = listMap(r, fn); then f_1 :: r_1;
  end matchcontinue;
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
  input list<Type_a> inVtype_alist;
  input FuncType inFunc;
  input Type_b inType_b;
  output Type_b outType_b;
public
  replaceable type Type_a subtypeof Any;
  replaceable type Type_b subtypeof Any;
  function FuncType
    input Type_a inType_a;
    input Type_b inType_b;
    output Type_b outType_b;
  end FuncType;
algorithm
  outType_b:=
  matchcontinue (inVtype_alist,inFunc,inType_b)
    local
      FuncType r;
      Type_b b,b_1,b_2;
      Type_a l;
      list<Type_a> lst;
    case ({},r,b) then b;
    case (l :: lst,r,b)
```

```
    equation
      b_1 = r(l, b);
      b_2 = listFold(lst, r, b_1); then b_2;
  end matchcontinue;
end listFold;
```

6.4   Exercises

See Appendix D.

Appendix A

MetaModelica Grammar

This appendix contains the grammar of the MetaModelica language. This is the grammar for the MetaModelica 1.0 extended subset only, not including standard Modelica.

Below is brief description of the MetaModelica concrete syntax. Keywords and special symbols (eg, *, &, =) are shown in bold letters, other tokens are shown in capital letters.

```
modelica
        : stored_definition
                ( stored_definition )

(* PATTERNS *)
pat     (* patterns possibly starting with LPAREN *)
        : ident EQ pat
        | ident AS pat
        | pat_a

pat_a
        : pat_b COLONCOLON pat_a
        | pat_b

pat_b   (* simple patterns possibly starting with LPAREN *)
        : LPAREN RPAREN
        | LPAREN pat RPAREN
        | LPAREN pat COMMA pat_comma_plus RPAREN
        | pat_d

pat_c   (* patterns not starting with LPAREN *)
        : pat_d COLONCOLON pat_c
        | pat_d

pat_d   (* simple patterns not starting with LPAREN *)
        : name_path pat_star
        | name_path pat_e
        | pat_e

pat_e   (* atomic patterns not starting with LPAREN *)
        : WILD
        | SUB_INT ICON %prec UNARY
        | SUB_REAL RCON %prec UNARY
        | ADD_INT ICON %prec UNARY
        | ADD_REAL RCON %prec UNARY
        | ICON
        | RCON
        | SCON
        | name_path
        | FALSE
        | TRUE
        | LBRACK pat_comma_star RBRACK
        | LBRACE pat_comma_star RBRACE

seq_pat
        : (*empty*)
        | pat_c          (* cannot start with LPAREN *)
        | pat_star
        | ident AS pat
        | ident EQ pat

pat_star
        : LPAREN pat_comma_star RPAREN

pat_comma_star
        : (*empty*)
        | pat_comma_plus
```

```
pat_comma_plus
        : pat
        | pat COMMA pat_comma_plus

(* SHORT IDENTIFIERS *)
ident
        : IDENT

stored_definition
    : class_type ident opt_string_comment composition END ident SEMICOLON

class_definition:
        class_type ident class_specifier

class_type:
        RECORD
|       TYPE
|       PACKAGE
|       PARTIAL FUNCTION
|       FUNCTION
|       UNIONTYPE


class_specifier:
        opt_string_comment composition END ident
        | opt_string_comment
        | EQ type_specifier comment
        | EQ enumeration

enumeration:
        ENUMERATION LPAREN enum_list RPAREN comment

enum_list:
        enumeration_literal COMMA enum_list
        | enumeration_literal

enumeration_literal:
        ident

variability_prefix:
        PARAMETER
|       CONSTANT
|       (* empty *)

direction_prefix:
        INPUT
|       OUTPUT
|       (* empty *)


component_clause:
        direction_prefix type_specifier component_list
        | CONSTANT direction_prefix type_specifier component_list
        | direction_prefix FUNCTION type_specifier component_list
        | direction_prefix REPLACEABLE FUNCTION component_list EXTENDS type_specifier


component_clause1:
        variability_prefix direction_prefix
        type_specifier component_declaration          (* 063 *)

import_clause:
        IMPORT explicit_import_name comment           (* 064 *)
        | IMPORT implicit_import_name comment         (* 065 *)

explicit_import_name:
        ident EQ name_path                            (* 066 *)


implicit_import_name:
        name_path DOTSTAR                             (* 067 *)
        | name_path                                   (* 068 *)


composition:
        element_list                                  (* 069 *)
```

```
            | element_list composition_sublist              (* 070 *)

composition_sublist:
            PUBLIC element_list composition_sublist          (* 071 *)
          | PROTECTED element_list composition_sublist       (* 072 *)
          | LOCAL element_list composition_sublist (* 073 *)
          | algorithm_clause composition_sublist             (* 074 *)
          | equation_clause composition_sublist              (* 075 *)
          | external_clause SEMICOLON                        (* 076 *)
          | (* empty *)                                      (* 077 *)

external_clause:
            EXTERNAL external_function_call                  (* 078 *)

language_specification:
            SCON                                             (* 079 *)

external_function_call:
            language_specification                           (* 080 *)
          | language_specification component_reference EQ ident LPAREN opt_expression_list
RPAREN

opt_expression_list:
            expression_list                                 (* 082 *)
          | (* empty *)                                     (* 083 *)

element_list:
            element SEMICOLON element_list                  (* 084 *)
          | (* empty *)                                     (* 085 *)

element:
            component_clause                                (* 086 *)
          | REPLACEABLE class_or_component                  (* 087 *)
          | import_clause                                   (* 088 *)
          | extends_clause                                  (* 089 *)
          | class_definition                                (* 090 *)

class_or_component:
                component_clause                             (* 091 *)
          | class_definition                                (* 092 *)


subscript:
            expression                                      (* 093 *)
          | COLON                                           (* 094 *)

array_subscripts:
            LBRACK subscript RBRACK                         (* 095 *)
          | (* empty *)                                     (* 096 *)

type_specifier_list:
            type_specifier                                  (* 097 *)
          | type_specifier COMMA type_specifier_list        (* 098 *)


type_specifier:
            name_path                                       (* 099 *)
       | name_path LT_INT type_specifier_list GT_INT        (* 100 *)
       | type_specifier LBRACK COLON RBRACK                 (* 101 *)


component_list:
            component_declaration                           (* 102 *)
          | component_declaration COMMA component_list      (* 103 *)

component_declaration:
            declaration comment                             (* 104 *)


declaration:
            ident opt_modification                          (* 105 *)

modification:
            class_modification EQ expression                (* 106 *)
          | class_modification                              (* 107 *)
          | EQ expression                                   (* 108 *)
```

```
        | ASSIGN expression                              (* 109 *)


opt_modification:
        modification                                    (* 110 *)
        | (* empty *)                                   (* 111 *)

class_modification:
        LPAREN argument_list RPAREN                     (* 112 *)
        | LPAREN RPAREN                                 (* 113 *)

argument_list:
         argument COMMA argument_list                   (* 114 *)
        | argument                                      (* 115 *)


argument:
        optEACH optFINAL component_reference opt_modification opt_string_comment
        | optEACH optFINAL component_reference opt_modification
        | REDECLARE optEACH optFINAL REPLACEABLE class_definition opt_constraining_clause
        | REDECLARE optEACH optFINAL REPLACEABLE component_clause1 opt_constraining_clause
        | REDECLARE optEACH optFINAL class_definition      (* 120 *)
        | REDECLARE optEACH optFINAL component_clause1     (* 121 *)

equation_clause:
        EQUATION equation_annotation_list               (* 122 *)

equation_annotation_list:
         equation SEMICOLON equation_annotation_list    (* 123 *)
        | (* empty *)                                   (* 124 *)

constraining_clause:
        extends_clause                                  (* 125 *)


opt_constraining_clause:
        constraining_clause                             (* 126 *)
        | (* empty *)                                   (* 127 *)

extends_clause:
        EXTENDS name_path class_modification            (* 128 *)


algorithm_clause:
        ALGORITHM algorithm_annotation_list             (* 129 *)

algorithm_annotation_list:
        algorithm SEMICOLON algorithm_annotation_list   (* 130 *)
        | (* empty *)                                   (* 131 *)

equation:
        simple_expression eq_equals                     (* 132 *)
        | conditional_equation_e comment                (* 133 *)
        | for_clause_e comment                          (* 134 *)
        | connect_clause comment                        (* 135 *)
        | when_clause_e comment                         (* 136 *)
        | FAILURE LPAREN equation RPAREN comment        (* 137 *)
        | EQUALITY LPAREN equation RPAREN comment       (* 138 *)


eq_equals:
        EQ expression comment                       (* 139 *)
  | comment                                 (* 140 *)

alg_assign:
         ASSIGN expression comment                      (* 141 *)
  | comment                                 (* 142 *)


algorithm:
        simple_expression alg_assign                    (* 143 *)
        | conditional_equation_a comment                (* 144 *)
        | for_clause_a comment                          (* 145 *)
        | while_clause comment                          (* 146 *)
        | when_clause_a comment                         (* 147 *)
        | FAILURE LPAREN algorithm RPAREN comment       (* 148 *)
```

```
        | EQUALITY LPAREN algorithm RPAREN comment        (* 149 *)

equation_elseif:
        ELSEIF expression THEN equation_list equation_elseif
        | (* empty *)                                     (* 151 *)


algorithm_elseif:
        ELSEIF expression THEN algorithm_list algorithm_elseif
        | (* empty *)                                     (* 153 *)

opt_equation_else:
        ELSE equation_list                                (* 154 *)
        | (* empty *)                                     (* 155 *)

opt_algorithm_else:
        ELSE algorithm_list                               (* 156 *)
        | (* empty *)                                     (* 157 *)


conditional_equation_e:
        IF expression THEN equation_list equation_elseif opt_equation_else END IF


conditional_equation_a:
        IF expression THEN algorithm_list algorithm_elseif opt_algorithm_else END IF

for_indices:
        for_indice COMMA for_indices                      (* 160 *)
        | for_indice                                      (* 161 *)

for_indice:
        ident                                             (* 162 *)
        | ident IN expression                             (* 163 *)

for_clause_e:
        FOR for_indices LOOP equation_list END FOR        (* 164 *)

for_clause_a:
        FOR for_indices LOOP algorithm_list END FOR       (* 165 *)


while_clause:
        WHILE expression LOOP algorithm_list END WHILE    (* 166 *)


when_clause_e:
        WHEN expression THEN equation_list else_when_e END WHEN

else_when_e:
        ELSEWHEN expression THEN equation_list else_when_e (* 168 *)
        | (* empty *)                                     (* 169 *)

when_clause_a:
        WHEN expression THEN algorithm_list else_when_a END WHEN


else_when_a:
        ELSEWHEN expression THEN algorithm_list else_when_a(* 171 *)
        | (* empty *)                                     (* 172 *)


equation_list:
        equation SEMICOLON equation_list                  (* 173 *)
        | (* empty *)                                     (* 174 *)


algorithm_list:
        algorithm SEMICOLON algorithm_list                (* 175 *)
        | (* empty *)                                     (* 176 *)


connect_clause:
        CONNECT LPAREN component_reference COMMA component_reference RPAREN

local_element_list:
```

```
      LOCAL element_list                              (* 178 *)
      | (* empty *)                                   (* 179 *)

match:
      MATCH                                           (* 180 *)
      | MATCHCONTINUE                                 (* 181 *)

match_expression:
  match expression opt_string_comment
      local_element_list
      case_list
      case_else
  END match                                     (* 182 *)

case_list:
       case_stmt case_list                            (* 183 *)
      | case_stmt                                      (* 184 *)

equation_clause_case:
      EQUATION equation_annotation_list         (* 185 *)
      | (* empty *)                             (* 186 *)

case_stmt:
       CASE seq_pat opt_string_comment local_element_list equation_clause_case THEN
expression SEMICOLON

case_else:
  (* empty *)                                    (* 188 *)
| ELSE opt_string_comment local_element_list equation_clause_case THEN expression SEMICOLON


expression:
      ident AS expression                        (* 190 *)
|   LPAREN ident AS expression RPAREN        (* 191 *)
    | if_expression                          (* 192 *)
      | simple_expression                       (* 193 *)
      | match_expression                        (* 194 *)

if_expression:
      IF expression THEN expression elseif_expression_list ELSE expression

elseif_expression_list:
      ELSEIF expression THEN expression elseif_expression_list
      | (* empty *)                             (* 197 *)

simple_expression:
      logical_expression                         (* 198 *)
      | logical_expression COLONCOLON simple_expression  (* 199 *)
      | logical_expression COLON logical_expression    (* 200 *)
      | logical_expression COLON logical_expression COLON logical_expression

logical_expression:
       logical_term                              (* 202 *)
      | logical_term OR logical_expression       (* 203 *)

logical_term:
      logical_factor                             (* 204 *)
      | logical_factor AND logical_term          (* 205 *)

logical_factor:
      relation                                   (* 206 *)
      | NOT relation %prec UNARY                 (* 207 *)

relation:
      arithmetic_expression                      (* 208 *)
      | arithmetic_expression rel_op arithmetic_expression
      | LPAREN arithmetic_expression rel_op arithmetic_expression RPAREN


rel_op:
(* integer operators *)
      LT_INT                                     (* 211 *)
|     LE_INT                                     (* 212 *)
|     GT_INT                                     (* 213 *)
|     GE_INT                                     (* 214 *)
|     EQEQ_INT                                   (* 215 *)
```

```
|       NE_INT                                      (* 216 *)
 (* real operators *)
|       LT_REAL                                     (* 217 *)
|       LE_REAL                                     (* 218 *)
|       GT_REAL                                     (* 219 *)
|       GE_REAL                                     (* 220 *)
|       EQEQ_REAL                                   (* 221 *)
|       NE_REAL                                     (* 222 *)
 (* string operators *)
|   EQEQ_STRING                               (* 223 *)


addsub_op:
        ADD_INT                               (* 224 *)
|       ADD_REAL                              (* 225 *)
|       SUB_INT                               (* 226 *)
|       SUB_REAL                              (* 227 *)
|       ADD_STRING                            (* 228 *)
|       ADD_LIST                              (* 229 *)

arithmetic_expression2:
            term addsub_op arithmetic_expression2 (* 230 *)
        |   term
            ( term )

arithmetic_expression:
        unary_arithmetic_expression
        | unary_arithmetic_expression addsub_op arithmetic_expression2

unary_arithmetic_expression:
        term                                      (* 233 *)
        | ADD_INT term %prec UNARY                (* 234 *)
        | ADD_REAL term %prec UNARY               (* 235 *)
        | SUB_INT term %prec UNARY                (* 236 *)
        | SUB_REAL term %prec UNARY               (* 237 *)

muldiv_op:
        STAR                                      (* 238 *)
|   MUL_REAL                              (* 239 *)
|       DIV_INT                               (* 240 *)
|   DIV_REAL                              (* 241 *)

term:
        factor                                    (* 242 *)
        | factor muldiv_op term                   (* 243 *)

pow_op:
        POW_REAL                                  (* 244 *)

factor:
        primary                                   (* 245 *)
        | primary pow_op primary                  (* 246 *)

expression_matrix_list:
        expression_list SEMICOLON expression_matrix_list   (* 247 *)
        | expression_list                        (* 248 *)


expression_matrix:
        expression_matrix_list                    (* 249 *)

primary:
        ICON                                      (* 250 *)
        | RCON                                    (* 251 *)
        | SCON                                    (* 252 *)
        | FALSE                                   (* 253 *)
        | TRUE                                    (* 254 *)
        | WILD                                    (* 255 *)
        | component_reference LPAREN function_arguments RPAREN
        (* lists *)
        | LIST LPAREN function_arguments RPAREN            (* 257 *)
        | LBRACE function_arguments RBRACE        (* 258 *)
        | LBRACE RBRACE                           (* 259 *)
        | FAIL LPAREN RPAREN                  (* 260 *)
        | LPAREN RPAREN                                 (* 261 *)
        | component_reference LPAREN RPAREN       (* 262 *)
        | component_reference                     (* 263 *)
```

```
        | LPAREN function_arguments RPAREN              (* 264 *)
        | LBRACK expression_matrix RBRACK              (* 265 *)


dot_name_path:
        DOT name_path                                  (* 266 *)
        | (* empty *)                                  (* 267 *)


name_path:
        ident dot_name_path                            (* 268 *)
        | TUPLE                                         (* 269 *)
        | LIST                                          (* 270 *)


component_reference:
        ident array_subscripts                         (* 271 *)
        | ident DOT component_reference                (* 272 *)


function_arguments:
        named_arguments                                (* 273 *)



named_arguments:
        named_argument COMMA named_arguments           (* 274 *)
        | named_argument                               (* 275 *)


named_argument:
        ident EQ expression                            (* 276 *)
        | expression                                   (* 277 *)


expression_list:
        expression                                     (* 278 *)
        | expression_list COMMA expression             (* 279 *)


comment:
        string_comment annotation                      (* 280 *)
        | string_comment                               (* 281 *)
        | annotation                                   (* 282 *)
        | (* empty *)                                  (* 283 *)


string_comment_add:
        ADD_INT string_comment                         (* 284 *)


string_comment:
        SCON                                           (* 285 *)
        | SCON string_comment_add                      (* 286 *)


opt_string_comment:
        string_comment                                 (* 287 *)
        | (* empty *)                                  (* 288 *)


annotation:
        ANNOTATION class_modification                  (* 289 *)
```

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 (This is the same as Table 5-1).

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}`
real power of	`^.`	`x ^. 2`
integer multiplicative	`*  /`	`2*3   2/3`
real multiplicative	`*.  /.`	`2.1 *. 3.2`
integer additive	`+  -  `+*expr* -*expr*	`a+b,  a-b,  +a,  -a`
real additive	`+. -.  +.` *expr* `-.` *expr*	`a+.b,  a-.b,  +.a,  -.a`
integer relational	`<  <=  >  >=  ==  <>`	`a<b,  a<=b,  a>b,  ...`
real relational	`<.  <=.  >.  >=.  ==.  <>.`	`a<.b, a<=.b, a>.b, ...`
string equality	`==&`	`str1 ==& str2`
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, `equality(expr1 = expr2)`, giving fail or succeed, which can be applied to values of primitive data types as well as to values of structured types such as arrays, lists, and trees.

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` − 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)` = $i1 \times i2$ 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 $i2 \neq 0$ 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 $i2 \neq 0$ 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 $i1 \geq i2$, otherwise `i2`.
- `intMin(i1,i2)` = $i1$ if $i1 \leq i2$, otherwise `i2`.
- `intLt(i1,i2)` = `true` if $i1 < i2$, otherwise `false`.
- `intLe(i1,i2)` = `true` if $i1 \leq i2$, otherwise `false`.
- `intEq(i1,i2)` = `true` if $i1 = i2$, otherwise `false`.
- `intNe(i1,i2)` = `true` if $i1 \neq i2$, otherwise `false`.
- `intGe(i1,i2)` = `true` if $i1 \geq i2$, 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)` = $r1 \times r2$.
- `realDiv(r1,r2)` = $r1 / r2$.
- `realMod(r1,r2)` = returns the integer modulus of $r1 / r2$. This is the value $r1 - i \times r2$, 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 $\geq$ r2, otherwise r2.
- `realMin(r1,r2)` = r1 if r1 $\leq$ r2, otherwise r2.
- `realLt(r1,r2)` = true if r1 < r2, otherwise `false`.
- `realLe(r1,r2)` = true if r1 $\leq$ r2, otherwise `false`.
- `realEq(r1,r2)` = true if r1 = r2, otherwise `false`.
- `realNe(r1,r2)` = true if r1 $\neq$ r2, otherwise `false`.
- `realGe(r1,r2)` = true if r1 $\geq$ 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  "Builtin list uniontype"
  record nil end nil;
  record cons
    replaceable type TypeVar subtypeof Any;
    TypeVar hd "The head of the list";
    list<TypeVar> tail "The rest of the list";
  end cons;
end list;

uniontype option  "Builtin option uniontype"
  record NONE end NONE;
  record SOME
    replaceable type TypeVar subtypeof Any;
    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 := matchcontinue (b1, b2)
     case (true, true) then true;
     case (_, _) then false;
end boolAnd;

function boolOr "Boolean or"
  input Boolean b1;
  input Boolean b2;
  output Boolean result;
algorithm
  result := matchcontinue (b1, b2)
    case (true, _) then true;
    case (_, true) then true;
    case (_, _) then false;
  end matchcontinue;
end boolOr;

function boolNot "Boolean not"
   input Boolean b;
   output Boolean result;
algorithm
  result := matchcontinue (b)
    case (true)  then false;
    case (false) then true;
  end matchcontinue;
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 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 "Appends two lists and returns the result"
  replaceable type TypeVar subtypeof Any;
  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 "Reverse the order of elements in the list"
  replaceable type TypeVar subtypeof Any;
  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 "Return the length of the list"
  replaceable type TypeVar subtypeof Any;
  input list<TypeVar> lst;
  output Integer result;
algorithm
  /* count the elements in the list and return the result */
end listLength;

function listMember "Verify if an element is part of the list"
  replaceable type TypeVar subtypeof Any;
  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 "Return the element of the list at the given index.
                  The index starts from 1."
  replaceable type TypeVar subtypeof Any;
  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 "Return a new list without the element at the
      given index. The index starts from 1."
  replaceable type TypeVar subtypeof Any;
  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   Array Operations

The `arrayGet` function is equivalent to the array indexing operator:

```
arrayGet(arr, index)  <=>  arr[index]
```

The standard Modelica `fill` function is equivalent to `arrayCreate`:

```
newarr = fill(v, n);  <=>  newarr = arrayCreate(v, n);
```

The following builtin array operations are available:

```
function arrayLength "Returns the length of the array"
  replaceable type TypeVar subtypeof Any;
  input TypeVar[:] arr;
  output Integer result;
algorithm
 /* Returns the length of the array */
end arrayLength;

function arrayGet "Returns the element of the array at the given index.
                   The index starts at 1. "
  replaceable type TypeVar subtypeof Any;
  input TypeVar[:] arr;
  input Integer index;
  output TypeVar result;
algorithm
 result := arr[index];
end arrayGet;

function arrayList "Returns the elements of the array as a list"
  replaceable type TypeVar subtypeof Any;
  input TypeVar[:] arr;
  output list<TypeVar> result;
algorithm
 /* return the elements of the array as a list */
end arrayList;

function listArray "Returns the elements of the list as an array"
  replaceable type TypeVar subtypeof Any;
  input list<TypeVar> lst;
  output TypeVar[:] result;
algorithm
 /* return the elements of the list as an array */
end listArray;
function arrayUpdate "Updates the array in place with a new element at
                      the given index. The index starts at 1."
  replaceable type TypeVar subtypeof Any;
  input TypeVar[:] arr;
  input Integer index;
  input TypeVar element;
  output TypeVar[:] result;
algorithm
  arr[index] := element;
end arrayUpdate;

function arrayCreate "Creates an array of specifed size with all
                      the elements intialized with specified element"
  replaceable type TypeVar subtypeof Any;
  input Integer size;
  input TypeVar element;
  output TypeVar[:] result;
algorithm
  /* create an array of specified size with all
     elements intialized to eleement*/
end arrayCreate;

function arrayCopy  "Create a copy of the specifed array"
  replaceable type TypeVar subtypeof Any;
  input TypeVar[:] arrInput;
```

```
    output TypeVar[:] arrOutput;
algorithm
  /* create a new array identical with the input and return it as output */
end arrayCopy;

function arrayAdd "Add an element at the end of the array"
  replaceable type TypeVar subtypeof Any;
  input TypeVar[:] arrInput;
  input TypeVar element;
  output TypeVar[:] arrOutput;
algorithm
  /* create a new array identical with the input add the element
   at the end and return it as output */
end arrayAdd;
```

B.3.9   If expressions

The `if_exp` function below is equivalent to if expressions: `if cond then exp1 else exp2;`.

```
function if_exp "select one of the inputs depending on the condtion"
  replaceable type TypeVar subtypeof Any;
  input Boolean condition;
  input TypeVar input;
  input TypeVar input2;
  output TypeVar output1;
algorithm
  output1 := if condition then input1 else input2;
end if_exp;
```

B.3.10   Logical Variables

Note: Logical variables is an extension that may not be supported in future versions of MetaModelica.

Logical variables are specified with the keyword `lvar`. For example:: `lvar<String>` declares a logical variable that points to a `String` type element. Logical variables, as well as ordinary local variables, become unbounded when backtracking occurs and their values are set back to the value they had before the call that exited with failure. The builtin functions handling logical variables are described below:

```
function lvarNew "Create a new logical variable"
  replaceable type TypeVar subtypeof Any;
  output lvar<TypeVar> output1;
algorithm
  /* create a new logical variable and return it */
end lvarNew;

function lvarSet "Set a value in the logical variable"
  replaceable type TypeVar subtypeof Any;
  input lvar<TypeVar> logicalVariable;
  input TypeVar valueOfLogicalVariable;
algorithm
  /* set the value of the logical variable */
end lvarSet;

function lvarGet "Get the value from a logical variable"
  replaceable type TypeVar subtypeof Any;
  input lvar<TypeVar> logicalVariable;
  output Option(TypeVar) valueOfLogicaVariable;
algorithm
  /* return SOME(value) or NONE if logicalVariable was not set. */
end lvarGet;
```

B.3.11   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. The functions have been sorted in a bottom-up fashion, definition-before-use, even though that is not necessary in Modelica. Auxiliary utility functions and low level constructs appear first, whereas statements appear last since they directly or indirectly refer to all the rest.

```
package Pam

   "In this version the State is (environment, input stream, output stream).
   However, the passed I/O streams are not used and updated, instead the I/O
   is done through operating system calls.
     Input is done through the function read which just calls a C function
   doing a call to scanf. This works well if no backtracking occurs,
   as when print is used."

/* 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;



/* 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;
```

C.1.1   Statement evaluation

```
/*************** Statement evaluation **************/

  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;
        IdentList rest;
      case (env,ASSIGN(id,e1))                          // Assignment
        equation
          v1 = eval(env, e1);
          env2 = update(env, id, v1);  then env2;
      case (state1 as (env,istream,ostream), IF(comp,s1,s2))  // if true ...
        equation
          BOOLval(true) = eval(env, comp);
          state2 = evalStmt(state1, s1);  then state2;
      case (state1 as (env,istream,ostream),IF(comp,s1,s2))   // if false ...
        equation
          BOOLval(false) = eval(env, comp);
          state2 = evalStmt(state1, s2); then state2;
      case (state,WHILE(comp,s1))                        // while ...
        equation
          state2 = evalStmt(state, IF(comp,SEQ(s1,WHILE(comp,s1)),SKIP()));
        then state2;
      case (state as (env,istream,ostream), TODO(e1,s1))     // to e1 do s1 ..
        equation
          INTval(n1) = eval(env, e1);
          state2 = repeatEval(state, n1, s1); then state2;
      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;
      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;
      case (state,SEQ(stmt1,stmt2))                                // stmt1 ; stmt2
        equation
          state2 = evalStmt(state, stmt1);
          state3 = evalStmt(state2, stmt2); then state3;
      case (state,SKIP()) then state;                              // ; empty statement
    end matchcontinue;
  end evalStmt;
```

C.1.2   Expression Evaluation

```
/*************** Expression evaluation **************/

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;
    case (_,INT(v)) then INTval(v);                        // Integer constant v
    case (env,IDENT(id))
      equation
        failure(v = lookup(env,id));
        error("Undefined identifier", id);                 // If id not declared,
      then v;
    case (env,IDENT(id))
      equation
        v = lookup(env,id);                                // Value of identifier id
      then v;
    case (env,BINARY(e1,binop,e2)) equation               // expr1 binop expr2
      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
      local Boolean v3; equation
       INTval(v1) = eval(env, e1);
       INTval(v2) = eval(env, e2);
       v3 = applyRelop(relop, v1, v2); then BOOLval(v3);
  end matchcontinue;
end eval;
```

C.1.3   Arithmetic and Relational Operators

```
/*************** Arithmetic and relational operators **************/

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;
```

C.1.4 Auxiliary Utility Functions

```
/***************** Auxiliary utility functions *****************/

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 is hopefully in rest of list
    end matchcontinue;
end lookup;


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;

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;

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;

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;

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;
```

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, parsing, and Makefile files.

- Main
- ScanParse
- Absyn
- Env
- Eval
- lexer.l
- parser.y
- scanparse.c
- 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;

function main
  input StringLst inStringLst;
  type StringLst = list<String>;
protected
  Absyn.Prog ast;
algorithm
  ast := ScanParse.scanparse();
  ast := Eval.evalprog(ast);
  //?? should really call mainprogram recursively to have a loop
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

import Absyn;

function scanparse
  output Absyn.Prog outProg;

  external "C" ;
end scanparse;

end ScanParse;
```

C.2.3   PAMDECL Absyn

The Absyn package specifies the abstract syntax used by the rest of the specification, i.e., the other packages.

```
package Absyn  "Package for abstract syntax of PamDecl"

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

```
type Ident = String;

uniontype Expr
  record INTCONST   Integer int;   end INTCONST;
  record REALCONST  Real real;   end REALCONST;
  record BINARY  Expr exp1;  BinOp binOp;  Expr exp2;  end BINARY;
  record UNARY  UnOp unOp;  Expr exp;   end UNARY;
  record RELATION  Expr exp1;  RelOp relOp;  Expr exp2;  end RELATION;
  record VARIABLE   Ident ident;   end VARIABLE;
end Expr;

type StmtList = 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 expr;  StmtList stmtList1;  StmtList stmtList2;   end IF;
  record WHILE   Expr expr;  StmtList stmtList;   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.2.4   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  "Package for Environment types and functions of PamDecl"

type Ident = String;

uniontype Value  "Three types of values can be handled by the semantics"
  record INTVAL   Integer int;   end INTVAL;
  record REALVAL   Real real;   end REALVAL;
  record BOOLVAL   Boolean boolean;   end BOOLVAL;
end Value;

uniontype Value2  "Values for real-integer type lattice conversions"
  record INTVAL2  Integer int1;  Integer int2;   end INTVAL2;
  record REALVAL2  Real real1;  Real real2;   end REALVAL2;
end Value2;

uniontype Type  "Three kinds of types can be declared"
  record INTTYPE   end INTTYPE;
  record REALTYPE   end REALTYPE;
  record BOOLTYPE   end BOOLTYPE;
end Type;

uniontype Bind "Type for associating identifer, type, and value"
  record BIND   Ident id;  Type ty;  Value val;   end BIND;
end Bind;

type Env = list<Bind>;
```

```
// Initial environment of predefined constants false and true
constant Bind initial = list(
           BIND(("false",BOOLTYPE(),BOOLVAL(false))),
           BIND(("true",BOOLTYPE(),BOOLVAL(true))));


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 (BIND(id2,_,value) :: rest, id) then
      if id==&id2 then value   // id first in list
      else lookup(rest,id);    // id is hopefully in rest of list
  end matchcontinue;
end lookup;


function lookuptype  "lookuptype returns the type associated with an identifier.
                      If no association is present, lookuptype will fail."

  input Env inEnv;
  input Ident inIdent;
  output Type outType;
algorithm
  outType:=
  matchcontinue (inEnv,inIdent)
    local
      Ident id2,id;
      Type ty;  Env rest;
    case (BIND(id2,ty,_) :: rest, id) then
      if id==&id2 then ty          // id first in list
      else lookuptype(rest,id);   // id is hopefully in rest of list
  end matchcontinue;
end lookuptype;

function update  "update returns an updated environment containing a
                  typed variable-type-value association BIND(id,type,value)"
  input Env env;
  input Ident id;
  input Type ty;
  input Value value;
  output Env outEnv;
algorithm
  outEnv := BIND((id,ty,value) :: env)
end update;

end Env;
```

C.2.5 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

import Absyn;
import Env;

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(decls,stmts))
      equation
        env1 = Env.initial_();
        env2 = evalDeclList(env1, decls);
        env3 = evalStmtList(env2, stmts);
      then ();
  end matchcontinue;
end evalprog;




/* Evaluation of statements */

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
      type Env_BindLst = list<Env.Bind>;
      type Absyn_StmtLst = list<Absyn.Stmt>;
      Env.Value v,v2;
      Env.Type ty;
      Env_BindLst env1,env,env2;
      String id;
      Absyn.Expr e;
      Absyn_StmtLst c,a,ss;
    case (env,Absyn.ASSIGN(id,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(id,e))
      equation
        v = evalExpr(env, e);
        print("Error: assignment mismatch or variable missing\n"); then fail();
    case (env,Absyn.WRITE(e))
      equation
        v = evalExpr(env, e);
        printValue(v); then env;
    case (env,Absyn.NOOP()) then env;
    case (env,Absyn.IF(e,c,_))
      equation
        Env.BOOLVAL(true) = evalExpr(env, e);
        env1 = evalStmtList(env, c); then env1;
    case (env,Absyn.IF(e,_,a))
      equation
        Env.BOOLVAL(false) = evalExpr(env, e);
        env1 = evalStmtList(env, a); then env1;
    case (env,Absyn.WHILE(e,ss))
      equation
        Env.BOOLVAL(true) = evalExpr(env, e);
        env1 = evalStmtList(env, ss);
        env2 = evalStmt(env1, Absyn.WHILE(e,ss)); then env2;
    case (env,Absyn.WHILE(e,ss))
```

```
      equation
        Env.BOOLVAL(false) = evalExpr(env, e); then env;
    case (env,Absyn.IF(e,_,a))
      equation
        Env.BOOLVAL(false) = evalExpr(env, e);
        env1 = evalStmtList(env, a); then env1;
    case (env,Absyn.WHILE(e,ss))
      equation
        Env.BOOLVAL(true) = evalExpr(env, e);
        env1 = evalStmtList(env, ss);
        env2 = evalStmt(env1, Absyn.WHILE(e,ss)); then env2;
    case (env,Absyn.WHILE(e,ss))
      equation
        Env.BOOLVAL(false) = evalExpr(env, e); then env;
  end matchcontinue;
end evalStmt;


function evalStmtList "Evaluate a list of statements in an environment.
                        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;


/* Evaluation of Declarations */

function evalDecl "Evaluate a single declaration. 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(var,"integer"))
      equation
        env2 = Env.update(env, var, Env.INTTYPE(), Env.INTVAL(0)); then env2;
    case (env,Absyn.NAMEDECL(var,"real"))
      equation
        env2 = Env.update(env, var, Env.REALTYPE(), Env.REALVAL(0.0));
      then env2;
    case (env,Absyn.NAMEDECL(var,"boolean"))
      equation
        env2 = Env.update(env, var, Env.BOOLTYPE(), Env.BOOLVAL(false));
      then env2;
  end matchcontinue;
end evalDecl;

function evalDeclList "Evaluate a list of declarations,
                        extending the environment."
```

```
    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 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
      type Env_BindLst = list<Env.Bind>;
      Env_BindLst 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;
    case (env,Absyn.INTCONST(v)) then Env.INTVAL(v);
    case (env,Absyn.REALCONST(v))
      local Real v; then Env.REALVAL(v);
    case (env,Absyn.BINARY(e1,binop,e2)) "Binary operators"
      equation
        v1 = evalExpr(env, e1);
        v2 = evalExpr(env, e2);
        Env.INTVAL2(c1,c2) = binaryLub(v1, v2);
        v3 = applyIntBinary(binop, c1, c2); then Env.INTVAL(v3);
    case (env,Absyn.BINARY(e1,binop,e2))
      local Real c1,c2,v3;
      equation
        v1 = evalExpr(env, e1);
        v2 = evalExpr(env, e2);
        Env.REALVAL2(c1,c2) = binaryLub(v1, v2);
        v3 = applyRealBinary(binop, c1, c2); then Env.REALVAL(v3);
    case (_,Absyn.BINARY(_,_,_))
      equation
        print("Error: binary operator applied to invalid type(s)\n");
      then fail();
    case (env,Absyn.UNARY(unop,e1))    // Unary operators
      local Integer v1,v2;
      equation
        Env.INTVAL(v1) = evalExpr(env, e1);
        v2 = applyIntUnary(unop, v1); then Env.INTVAL(v2);
    case (env,Absyn.UNARY(unop,e1))
      local Real v1,v2;
      equation
        Env.REALVAL(v1) = evalExpr(env, e1);
```

```
          v2 = applyRealUnary(unop, v1); then Env.REALVAL(v2);
      case (_,Absyn.UNARY(_,_))
        equation
          print("Error: unary operator applied to invalid type\n");
        then fail();
      case (env,Absyn.RELATION(e1,relop,e2))  // Relation operators"
        local Boolean v3;
        equation
          v1 = evalExpr(env, e1);
          v2 = evalExpr(env, e2);
          Env.INTVAL2(c1,c2) = binaryLub(v1, v2);
          v3 = applyIntRelation(relop, c1, c2); then Env.BOOLVAL(v3);
      case (env,Absyn.RELATION(e1,relop,e2))
        local
          Real c1,c2;
          Boolean v3;
        equation
          v1 = evalExpr(env, e1);
          v2 = evalExpr(env, e2);
          Env.REALVAL2(c1,c2) = binaryLub(v1, v2);
          v3 = applyRealRelation(relop, c1, c2); then Env.BOOLVAL(v3);
      case (_,Absyn.RELATION(_,_,_))
        equation
          print("Error: relation operator applied to invalid type(s)\n");
        then fail();
      case (env,Absyn.VARIABLE(id))   // Variable lookup"
        local Env.Value v;
        equation
          v = Env.lookup(env, id); then v;
      case (env,Absyn.VARIABLE(id))
        equation
          failure(v = Env.lookup(env, id));
          print("Error: undefined variable (");
          print(id);
          print(")\n"); then fail();
    end matchcontinue;
  end evalExpr;


  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 c1,c2;
      case (Env.INTVAL(v1),Env.INTVAL(v2)) then Env.INTVAL2(v1,v2);
      case (Env.REALVAL(v1),Env.REALVAL(v2))
        local Real v1,v2; then Env.REALVAL2(v1,v2);
      case (Env.INTVAL(v1),Env.REALVAL(v2))
        local Real v2;
        equation
          c1 = intReal(v1); then Env.REALVAL2(c1,v2);
      case (Env.REALVAL(v1),Env.INTVAL(v2))
        local Real v1;
        equation
          c2 = intReal(v2); then Env.REALVAL2(v1,c2);
    end matchcontinue;
  end binaryLub;


  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 v2;
    case (Env.INTVAL(v),Env.INTTYPE()) then Env.INTVAL(v);
    case (Env.REALVAL(v),Env.REALTYPE())
      local Real v; then Env.REALVAL(v);
    case (Env.BOOLVAL(v),Env.BOOLTYPE())
      local Boolean v; then Env.BOOLVAL(v);
    case (Env.INTVAL(v),Env.REALTYPE())
      equation
        v2 = intReal(v); then Env.REALVAL(v2);
  end matchcontinue;
end promote;


/*  Auxiliary functions for applying the binary operators */


function applyIntBinary  "Apply integer 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 v1/v2;
  end matchcontinue;
end applyIntBinary;

function applyRealBinary  "Apply real binary operators"
  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;


/* Auxiliary functions for applying the unary operators */

function applyIntUnary  "Apply integer 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;

function applyRealUnary  "Apply unary real operators"
  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;



/* Auxiliary functions for applying the relational operators  */



function applyIntRelation  "Apply integer relational 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;

function applyRealRelation  "Apply real relational operators"
  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;


/* Evaluate the 'write' statement */

function printValue  "Evaluate the 'write' statement, i.e., print a value"
  input Env.Value inValue;
algorithm
  _:=
  matchcontinue (inValue)
    local
      String vstr;
      Integer v;
```

```
        case (Env.INTVAL(v))
          equation
            vstr = intString(v);
            print(vstr);
            print("\n"); then ();
        case Env.REALVAL(v)
          local Real v;
          equation
            vstr = realString(v);
            print(vstr);
            print("\n"); then ();
        case Env.BOOLVAL(true)
          equation
            print("true\n"); then ();
        case Env.BOOLVAL(false)
          equation
            print("false\n"); then ();
      end matchcontinue;
end printValue;


end Eval;
```

C.2.6   PAMDECL lexer.l

```
%{
#include <stdlib.h>
#include "parser.h"
#include "omc.h"
#include "yacclib.h"

#include "Absyn.h"

typedef void *omc_t;
extern omc_t yylval;

int absyn_integer(char *s);
int absyn_ident_or_keyword(char *s);

%}

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

%%

"("             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 Modelica 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=(omc_t) Absyn__INTCONST(mk_icon(atoi(s)));
  return T_CONST_INT;

}

/* Make an Modelica 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=(omc_t) Absyn__REALCONST(mk_rcon(atof(s)));
  return T_CONST_REAL;
}

/* Make an Modelica 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 = (omc_t) mk_scon(s);
  return T_IDENT;
}
```

C.2.7   PAMDECL parser.y

```
%{
#include <stdio.h>
#include "yacclib.h"
#include "Absyn.h"

typedef void *omc_t;
#define YYSTYPE omc_t
extern omc_t absyntree;


%}

%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 stmtList T_END T_PROGRAM
        { absyntree = Absyn__PROG($2,$4); }

decl_list
        :
            { $$ = mk_nil();}
        | decl decl_list
            { $$ = mk_cons($1,$2); }

decl
        : T_IDENT T_COLON T_IDENT T_SEMICOLON
        { $$ = Absyn__NAMEDECL($1,$3);}
```

```
stmtList
        :
            { $$ = mk_nil();}
        | stmt stmtList
            { $$ = 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 stmtList T_ELSE stmtList T_END T_IF
            { $$ = Absyn__IF($2,$4,$6); }
        | T_IF expr T_THEN stmtList T_END T_IF
            { $$ = Absyn__IF($2,$4,mk_cons(Absyn__NOOP,mk_nil())); }

while_stmt
        : T_WHILE expr T_DO stmtList 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);}

%%
```

C.2.8   PAMDECL scanparse.c

```
/* Glue to call parser (and thus scanner) from Modelica */

#include <stdio.h>
#include "omc.h"

/* Provide error reporting function for yacc */

yyerror(char *s)
{
  extern int yylineno;
  fprintf(stderr,"Error: bad syntax on line %d.\n",yylineno);
}

/* The yacc parser will deposit the syntax tree here */

void *absyntree;

/* No init for this package */

void ScanParse_5finit(void) {}

/* The glue function */

OMC_BEGIN_LABEL(ScanParse__scanparse)
{
  if (yyparse() !=0)

  {
    fprintf(stderr,"Fatal: parsing failed!\n");
    OMC_TAILCALLK(omcFC);
  }

  omcA0=absyntree;
  OMC_TAILCALLK(omcSC);
}
OMC_END_LABEL
```

C.2.9   PAMDECL Makefile

```
# Makefile for building PAMDECL
#
# ??Note: LDFLAGS, CFLAGS are non-portable for some Unix systems
```

```
# VARIABLES

SHELL = /bin/sh
LDLIBS = -lomc -ll # Order is essential; we want libomc main, not libll!
LDFLAGS = -L$(OMCRUNTIME)/lib/plain/
CC = gcc
CFLAGS = -I$(OMCRUNTIME)/include/plain/ -g -I..

# EVERYTHING
all:    pamdecl

# EXECUTABLE

COMMONOBJS=yacclib.o
VSLOBJS=main.o lexer.o parser.o scanparse.o absyn.o env.o eval.o

pamdecl: $(VSLOBJS) $(COMMONOBJS)
        $(CC) $(LDFLAGS) $(VSLOBJS) $(COMMONOBJS) $(LDLIBS) -o pamdecl

# MAIN ROUTINE WRITTEN IN Modelica NOW

main.o: main.c
main.c main.h: main.om
        mmc -c main.om

# YACCLIB

yacclib.o:  yacclib.c
        $(CC) $(CFLAGS) -c -o yacclib.o yacclib.c

# LEXER

lexer.o:  lexer.c parser.h absyn.h
lexer.c:  lexer.l
        lex -t lexer.l >lexer.c

# PARSER

parser.o:  parser.c absyn.h
parser.c parser.h:  parser.y
        yacc -d parser.y
        mv y.tab.c parser.c
        mv y.tab.h parser.h

# INTERFACE TO SCANNER/PARSER (Modelica CALLING C)


scanparse.o:  scanparse.c absyn.h

# ABSTRACT SYNTAX

absyn.o:  absyn.c
absyn.c absyn.h:  absyn.om
        mmc -c absyn.om

# ENVIRONMENTS

env.o:  env.c
env.c env.h:  env.om
        mmc -c env.om

# EVALUATION

eval.o:  eval.c
eval.c eval.h:  eval.om absyn.h env.h
        mmc -c eval.om
```

```
# AUX

clean:
        $(RM) pamdecl $(COMMONOBJS) $(VSLOBJS) main.c main.h lexer.c parser.c pa
rser.h absyn.c absyn.h env.c env.h eval.c eval.h *~
```

C.3 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, gram.y, Main.mo, Parse.mo, parse.c, yacclib.c, yacclib.h and makefile.

Theses files for MetaModelica 1.0 can be found in http://openmodelica.org/metamodelica/exercises/.

The executable is built by typing:

```
>> make pamtrans
```

C.3.1 lexer.l

```
%{
#include "gram.h"
#include "yacclib.h"
#include "omc.h"
#include "Absyn.h"

typedef void *omc_t;
extern omc_t yylval;

int absyn_integer(char *s);
int absyn_ident_or_keyword(char *s);

%}

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;


%%

/* Make an Modelica 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=(omc_t) Absyn__INT(mk_icon(atoi(s)));
  return T_INTCONST;
}

/* Make an Modelica 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 = (omc_t) mk_scon(s);
        return T_IDENT;
}
gram.y
%{
#include <stdio.h>
#include "yacclib.h"
#include "omc.h"
#include "Absyn.h"

typedef void *omc_t;
#define YYSTYPE omc_t

extern omc_t absyntree;

%}

%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
                            { $$ = mk_cons($1, mk_nil()); }
                    | variable variable_list
                            { $$ = 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.3.2   PAMTRANS Absyn.mo

```
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;
```

C.3.3 PAMTRANS Trans.mo

```
package Trans

import Absyn;
import Mcode;


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;


/*************** 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;

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

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

```
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;

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;

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;

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;


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);

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);

case Absyn.SEQ(stmt1,stmt2)             // stmt1 ; stmt2
  equation
```

```
        cod1 = transStmt(stmt1);
        cod2 = transStmt(stmt2);
    end matchcontinue;

end transStmt;


function transExpr  "Arithmetic expression translation"
  input Absyn.Exp inExp;
  output list<Mcode.MCode> outMCodeList;
algorithm
  outMCodeList:=
  matchcontinue (inExp)
    local
      Integer v;
      String id;
      list<Mcode.MCode> 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

    case Absyn.BINARY(e1,binop,e2) " Arith binop: simple case, expr2 is just an
                                    identifier or constant:   expr1 binop expr2 "
      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;


    case Absyn.BINARY(e1,binop,e2) "Arith binop: general case, expr2 is a more
                                   complicated expr:   expr1 binop expr2"
      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:=
  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 temporary"
  output Mcode.MOperand outMOperand;
protected
```

```
    Integer no;
algorithm
  no = tick();
  outMOperand := Mcode.T(no);
end gentemp;


function listAppend3
  input list<Type_a> l1;
  input list<Type_a> l2;
  input list<Type_a> l3;
  output list<Type_a> l13;
  replaceable type Type_a subtypeof Any;
protected
  list<Type_a> l12;
algorithm
  l12 := listAppend(l1, l2);
  l13 := listAppend(l12, l3);
end listAppend3;


function listAppend5
  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;
  output list<Type_a> l15;
  replaceable type Type_a subtypeof Any;
protected
  list<Type_a> l13;
algorithm
  l13 := listAppend3(l1, l2, l3);
  l15 := listAppend3(l13, l4, l5);
end listAppend5;



function listAppend6
  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;
  replaceable type Type_a subtypeof Any;
protected
  list<Type_a> l13,l46;
algorithm
  l13 := listAppend3(l1, l2, l3);
  l46 := listAppend3(l4, l5, l6);
  l16 := listAppend(l13, l46);
end listAppend6;



function listAppend10
  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;
  input list<Type_a> l7;
  input list<Type_a> l8;
  input list<Type_a> l9;
  input list<Type_a> l10;
  output list<Type_a> l110;
  replaceable type Type_a subtypeof Any;
protected
```

```
    list<Type_a> l15;
algorithm
  l15 := listAppend5(l1, l2, l3, l4, l5);
  l110 := listAppend6(l15, l6, l7, l8, l9, l10);
end listAppend10;




/*************** 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);  // Only 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:=
  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;
```

C.3.4   PAMTRANS Mcode.mo

```
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;
```

C.3.5   PAMTRANS Emit.mo

```
package Emit
/* Print out the Mcode in textual assembly format
 * Note: this is not really part of the specification of PAM semantics,
 * rather it is low-level code generation.
```

```
 */
import Mcode;

function emitAssembly  "Print an Mcode instruction"
  input list<Mcode.MCode> inMCodeList;
algorithm
  _ :=
  matchcontinue (inMCodeList)
    local
      Mcode.MCode instr;
      list<Mcode.MCode> rest;
    case ({}) then ();
    case (instr :: rest)
      equation
        emitInstr(instr);
        emitAssembly(rest); then ();
  end matchcontinue;
end emitAssembly;

function emitInstr
  input Mcode.MCode inMCode;
algorithm
  _ :=
  matchcontinue (inMCode)
    local
      String op;
      Mcode.MBinOp mbinop;
      Mcode.MOperand mopr,mlab;
      Mcode.MCondJmp jmpop;
    case (Mcode.MB(mbinop,mopr))  // Print an Mcode instruction
      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;

C.3.6 PAMTRANS Main.mo

```
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
  Absyn.Stmt program;
  list<Mcode.MCode> mcode;
algorithm
  program := Parse.parse();
  mcode := Trans.transProgram(program);
  Emit.emitAssembly(mcode);
end main;

end Main;
```

C.3.7 PAMTRANS Parse.mo

```
package Parse
  import Absyn;

function parse
  output Absyn.Stmt outStmt;

  external "C" ;
end parse;

end Parse;
```

C.3.8 PAMTRANS parse.c

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include "omc.h"

#ifndef OMC_INSPECTBOX
#define OMC_INSPECTBOX(d,h,p)
(OMC_ISIMM((d)=(p))?0:(((h)=(void*/OMC_GETHDR((p))),0))

#define omc_prim_deref_imm(x) x
#endif

void Parse_5finit(void) {}

void *absyntree;

OMC_BEGIN_LABEL(Parse__parse) {
    void *a0, *a0hdr;
    OMC_INSPECTBOX(a0, a0hdr, omcA0);
    if( a0hdr == OMC_IMMEDIATE(OMC_UNBOUNDHDR) )
        OMC_TAILCALLK(omcFC);
    else {
        if(yyparse()==0) {
            omcA0 = absyntree;
            OMC_TAILCALLK(omcSC);
        }
        else OMC_TAILCALLK(omcFC);
```

```
      }
}
makefile
# Makefile for building translational version of PAM
#
# ??Note: LDFLAGS, CFLAGS are non-portable for some Unix systems

# VARIABLES

SHELL = /bin/sh
LDLIBS = -lomc -ll # Order is essential; we want libomc main, not libll!
LDFLAGS = -L$(OMCRUNTIME)/lib/plain/
CC = gcc
CFLAGS = -I$(OMCRUNTIME)/include/plain/ -g -I..
MMC = $(OMCRUNTIME)/bin/mmc

# EVERYTHING
all:    pamtrans

# EXECUTABLE

COMMONOBJS=yacclib.o
VSLOBJS=main.o lexer.o gram.o parse.o absyn.o mcode.o trans.o emit.o

pamtrans: $(VSLOBJS) $(COMMONOBJS)
      $(CC) $(LDFLAGS) $(VSLOBJS) $(COMMONOBJS) $(LDLIBS) -o pamtrans

# MAIN ROUTINE WRITTEN IN Modelica NOW

main.o: main.c
main.c main.h: main.omc
      $(MMC) -c main.omc

# YACCLIB

yacclib.o:  yacclib.c
      $(CC) $(CFLAGS) -c -o yacclib.o yacclib.c

# LEXER

lexer.o:  lexer.c gram.h absyn.h
lexer.c:  lexer.l

      lex -t lexer.l >lexer.c

# PARSER

gram.o:  gram.c gram.h
gram.c gram.h:  gram.y
      yacc -d gram.y
      mv y.tab.c gram.c
      mv y.tab.h gram.h

# INTERFACE TO SCANNER/PARSER (Modelica CALLING C)

parse.o:  parse.c absyn.h

# ABSTRACT SYNTAX

absyn.o:  absyn.c
absyn.c absyn.h:  absyn.omc
      $(MMC) -c absyn.omc

# TRANSLATION

trans.o: trans.c
trans.c trans.h:  trans.omc absyn.h
```

```
        $(MMC) -c trans.omc

# EMISSION

emit.o:  emit.c
emit.c emit.h:  emit.omc
        $(MMC) -c emit.omc

# INTERMEDIATE FORM

mcode.o:  mcode.c
mcode.c mcode.h:  mcode.omc
        $(MMC) -c mcode.omc


# AUX

clean:
        $(RM) pamtrans $(COMMONOBJS) $(VSLOBJS) main.c main.h lexer.c parser.c
parser.h absyn.c absyn.h env.c env.h eval.c eval.h *~#include <stdlib.h>
```

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;
        else 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
  _ :=
  matchcontinue arg
    case (n_str::_)
      local
        Integer i, n;
        String str, 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")));

        // your code here -- uncomment these when you write the functions
        // 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 matchcontinue;
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, you only have to uncomment some cases in `parser.y`.

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 "file Exp1.mo"

uniontype Exp

  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;

  // your code here
  // add 2 new records called FACop and POWop

end Exp;

public function eval "Abstract syntax of the language Exp1:
                     Evaluation semantics  of Exp1"
  input Exp inExp;
  output Integer outInteger;
algorithm
  outInteger:=
  matchcontinue (inExp)
    local
      Integer ival,v1,v2;
      Exp e1,e2,e;
```

```
      case (INTconst(integer = ival))
        "eval of an integer node is the integer itself" then ival;
      case (ADDop(exp1 = e1,exp2 = e2))
        "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."
        equation
          v1 = eval(e1);
        then v1 + eval(e2);
      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 = 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;


%{
#include <stdio.h>
#include "yacclib.h"
#include "rml.h"
#include "exp1.h"

#define YYSTYPE rml_t

typedef void *rml_t;
extern rml_t absyntree;

%}

%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
                   { $$ = mk_box2(exp1__ADDop_3dBOX2,$1,$3);}
                 | expression  T_SUB  term
                   { $$ = mk_box2(exp1__SUBop_3dBOX2,$1,$3);}

term             : u_element
                 | term  T_MUL  u_element
                   { $$ = mk_box2(exp1__MULop_3dBOX2,$1,$3);}
                 | term  T_DIV  u_element
                   { $$ = mk_box2(exp1__DIVop_3dBOX2,$1,$3);}

u_element        : element
                 | T_SUB  element
                   { $$ = mk_box1(exp1__NEGop_3dBOX1,$2);}
                 /* uncomment here to have factorial and power operators
                 | T_FACTORIAL  element
                   { $$ = mk_box1(exp1__FACop_3dBOX1,$2);}
                 | element T_POW  u_element
                   { $$ = mk_box2(exp1__POWop_3dBOX2,$1,$3);}
                 */

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

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.3.

The following package, available in Exp2.mo, should be modified and extended:

```
package Exp2 "file Exp2.mo"

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;

  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 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 applyBinop(binop, eval(e1), eval(e2));
      case (UNARY(unOp = unop,exp = e))
        equation
          /*
          v1 = eval(e);
          v2 = applyUnop(unop, v1);
          */
        then applyUnop(unop, eval(e));
    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 v1/v2;
    end matchcontinue;
  end applyBinop;

  protected function applyUnop
    input UnOp inUnOp;
    input Integer inInteger;
    output Integer outInteger;
  algorithm
```

```
    outInteger:=
  matchcontinue (inUnOp,inInteger)
    local Integer v;
    case (NEG(),v) then -v;
  end matchcontinue;
 end applyUnop;

 end Exp2;
```

D.1.4   Exercise 03_Assignment – Printing AST and Environments

In this exercise you will add new functions for printing abstract syntax tress and environments:

- The assignment statements present in the current program in the Assignment language before the actual evaluation.
- The environment after it was augmented with the assignments
- Search for // your code here within Assignment.mo.

A solution is available in the file SOLUTION.txt and in Appendix E.4.

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 Assignment.mo where you should insert your code:

```
package Assignment "Assignment.mo"

type ExpLst = list<Exp>;

uniontype Program "Abstract syntax for the Assignments language"
  record PROGRAM
    ExpLst expLst;
    Exp exp;
  end PROGRAM;
end Program;

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;

  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;

public
type Value = Integer "Values stored in environments";

public
type VarBnd = tuple<Ident,Value> "Bindings and environments";

public
type Env = list<VarBnd>;

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;
    case ((id2,value) :: _,id)
      "lookup returns the value associated with an identifier.
       If no association is present, lookup will fail."
      equation
        equality(id = id2); then value;
    case ((id2,_) :: rest,id)
      equation
        failure(equality(id = id2));
        value = lookup(rest, id); then value;
  end matchcontinue;
end lookup;

/* lookup function using if expression
 * doesn't work in this MetaModelica version
 * because both then part and else part are evaluated
 * disregarding the condition
 */
/*
function lookup
  input Env inEnv;
  input Ident inIdent;
  output Value outInteger;
algorithm
 outInteger:=
  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(v = 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 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;

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 v1/v2;
  end matchcontinue;
end applyBinop;

protected function applyUnop
  input UnOp inUnOp;
  input Integer inInteger;
  output Integer outInteger;
algorithm
  outInteger:=
  matchcontinue (inUnOp,inInteger)
    local Value v;
    case (NEG(),v) then -v;
  end matchcontinue;
```

```
    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;
    case (e,{}) then e;
            // the environment stay the same if there are no expressions
    case (env,exp :: expl)
            // 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
      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:=
  matchcontinue (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 matchcontinue;
end evalprogram;


// your code here

end Assignment;
```

D.1.5   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, you only have to uncomment some grammar rules in parser.y

The solution is available in the file SOLUTION.txt and in Appendix E.5.

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"

type ExpLst = list<Exp>;

uniontype Program "Abstract syntax for the Assigntwotype language"
  record PROGRAM
    ExpLst expLst;
    Exp exp;
  end PROGRAM;
end Program;

uniontype Exp
  record INT
    Integer integer;
  end INT;

  record REAL
    Real real;
  end REAL;

  // your code here
  // add a record called STRING

  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;
```

```
public
uniontype Value "Values stored in environments"
  record INTval
    Integer integer;
  end INTval;

  record REALval
    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 x_1;
      Integer x;
    case (INTval(integer = x))
      equation
        x_1 = intString(x);
        print(x_1); then ();
    case (REALval(real = x))
      local Real x;
      equation
        x_1 = realString(x);
        print(x_1); 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); 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;
      String sval;
      Value value,v1,v2;
      Ident id;
      Exp e1,e2,e,exp;
      BinOp binop;
      UnOp unop;
    case (env,INT(integer = ival)) then (env,INTval(ival));
    case (env,REAL(real = rval)) then (env,REALval(rval));
    // your code here
    // case (env, STRING(...)) ...
    case (env,IDENT(ident = id)) "variable id"
      equation
        (env2,value) = lookupextend(env, id); then (env2,value);
    case (env,BINARY(exp1 = e1,binOp2 = binop,exp3 = e2)) "int binop int"
      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));
    case (env,BINARY(exp1 = e1,binOp2 = binop,exp3 = e2)) "int/real binop
int/real"
      local Real x,y,z;
      equation
        (env1,v1) = eval(env, e1);
        (env2,v2) = eval(env, e2);
        REAL2(real1 = x,real2 = y) = typeLub(v1, v2);
        z = applyRealBinop(binop, x, y);
      then (env2,REALval(z));
    case (env,UNARY(unOp = unop,exp = e)) "int unop exp"
```

```
      equation
        (env1,INTval(integer = x)) = eval(env, e);
        y = applyIntUnop(unop, x);
      then (env1,INTval(y));
    case (env,UNARY(unOp = unop,exp = e)) "real unop exp"
      local Real x,y;
      equation
        (env1,REALval(real = x)) = eval(env, e);
        y = applyRealUnop(unop, x);
      then (env1,REALval(y));
    case (env,ASSIGN(ident = id,exp = exp)) "eval of an assignment node returns
the updated environment and
    the assigned value id := 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 = y))
      local Real y;
      equation
        x2 = intReal(x); then REAL2(x2,y);
    case (REALval(real = x),INTval(integer = y))
      local Real x;
      equation
        y2 = intReal(y); then REAL2(x,y2);
    case (REALval(real = x),REALval(real = y))
      local Real x,y; then REAL2(x,y);
  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 x/y;
  end matchcontinue;
end applyIntBinop;


protected function applyRealBinop
  input BinOp inBinOp1;
  input Real inReal2;
  input Real inReal3;
  output Real outReal1;
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;
    case ((id2,value) :: _,id) "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."
      equation
        equality(id = id2); then value;
    case ((id2,_) :: rest,id) "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.
"
      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;
    case (env,id) "Return value of id in env. If id not present, add id and
return 0"
      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;


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.1.6   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.6 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.7.

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: add these functions to `Functions.mo` for printing.

```
// 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
  _ := matchcontinue(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 matchcontinue;
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
  _ := matchcontinue(oneRecordVar)
    local
      String cmp1;
      Types.Alias cmp2;
    case (Types.OneRecord(cmp1, cmp2))
      equation
        print("OneRecord(");
        print("\"" +& cmp1 +& "\"");
        print(", ");
        printAlias(cmp2);
        print(")");
      then ();
  end matchcontinue;
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;
    case (Types.FirstAlternative(cmp1, cmp2))
      equation
        print("FirstAlternative(");
        print("\"" +& cmp1 +& "\"");
        print(", ");
        print("\"" +& cmp2 +& "\"");
        print(")");
      then ();
    case (Types.SecondAlternative(cmp1, cmp2))
      local
        Types.Select cmp1, cmp2;
      equation
        print("SecondAlternative(");
        printSelect(cmp1);
        print(", ");
        printSelect(cmp2);
        print(")");
      then ();
    case (Types.ThirdAlternative(cmp1))
      local
        Types.Select cmp1;
      equation
        print("ThirdAlternative(");
        printSelect(cmp1);
        print(")");
      then ();
  end matchcontinue;
end printSelect;
```

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 02b_Exp2 – Adding New Features to a Small Language

No solution available – analogous to the solution of o2b_Exp2..

E.4   Solution 03_Assignment – Printing AST and Environment

The solution: Add the following functions to `Assignments.mo` and call them within the function evalprogram:

```
function printAssignments
  input ExpLst assignList;
algorithm
  _ := matchcontinue(assignList)
    local ExpLst expLst; Exp exp;
    case ({}) then (); // if nothing is in the list don't print anything
    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 exp;
algorithm
  _ := matchcontinue(exp)
    local
      Integer i;
      Exp exp1, exp2, exp;
      Ident id;
    case(INT(i))
      equation print(intString(i));
      then ();
    case(BINARY(exp1, op, exp2))
      local BinOp op;
      equation
        printExp(exp1);
        printBinaryOp(op);
        printExp(exp2);
      then ();
    case (UNARY(op, exp))
      local UnOp op;
      equation
        printUnaryOp(op);
        printExp(exp);
      then ();
    case(ASSIGN(id, exp))
      equation
        print(id);
        print(" = ");
        printExp(exp);
      then ();
    case(IDENT(id))
      equation print(id);
      then ();
  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
  _ := matchcontinue (op)
    case (NEG()) equation print("-"); then ();
  end matchcontinue;
end printUnaryOp;
//--------------

// your code here

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)::{})
      equation
        print(id);
        print(" = ");
        print(intString(val));
        print("\n");
      then ();
    case ((id, val)::varBndLstRest)
      equation
        print(id);
        print(" = ");
        print(intString(val));
        print(", ");
        printEnvironment(varBndLstRest);
      then ();
  end matchcontinue;
end printEnvironment;
```

E.5   Solution 04a_AssignTwoType – Adding a New Type to a Language

The solution includes the following changes in the file `AssignTwoType.mo`:

Exp type additions:

```
record STRING
  String str;
end STRING;
```

function eval additions:

```
    case (env,STRING(str = sval))
      equation
        z = stringInt(sval);
      then (env,INTval(z));
```

E.6   Solution 04b_ModAssignTwoType – Modularized Specification

Apply similar changes as in the previous exercise. A modularized AssignTwoType is available in the files Absyn.mo, Eval.mo, Main.mo, Parse.mo.

E.7   Solution 05_advanced – Polymorphic Types and Higher Order Functions

The solution is shown below:

```
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 = string_compare(s1, s2);
      then false;
    case (s1, s2)
      equation
        z = string_compare(s1, s2);
        true = (z < 0);
      then true;
    case (s1, s2)
      equation
        z = string_compare(s1, s2);
        false = (z < 0);
      then false;
  end matchcontinue;
end compareString;


function quicksort
  input  list<Type_a> inList;
  input  list<Type_a> accList;
  input  FuncType      comparator;
  output list<Type_a> outList;
public
  replaceable type Type_a subtypeof Any;
  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;
    case (x::l, acc, comparator)
      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
  input  Type_a inList;
  input  list<Type_a> accList;
  input FuncType comparator;
  output list<Type_a> outList1;
  output list<Type_a> outList2;
public
  replaceable type Type_a subtypeof Any;
  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;
```

```
      //function FuncType
      //  input Type_a el1;
      //  input Type_a el2;
      //  output Boolean cmp;
      //end FuncType;
      FuncType comparator;
    case (x, nil, _) then (nil, nil);
    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
  input  list<Type_a> inList;
  input FuncType comparator;
  output list<Type_a> outList;
public
  replaceable type Type_a subtypeof Any;
  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
  input  list<Type_a> inList;
```

```
    input FuncType f;
    output list<Type_b> outList;
public
  replaceable type Type_a subtypeof Any;
  replaceable type Type_b subtypeof Any;
  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 {};
    case (x::rest, f)
      equation
        y = f(x);
        lst = map1(rest, f);
      then y::lst;
  end matchcontinue;
end map1;

function map0
  input list<Type_a> inList;
  input FuncType f;
  output list<Type_a> outList;
public
  replaceable type Type_a subtypeof Any;
  function FuncType
    input  Type_a elIn;
  end FuncType;
algorithm
  outList := 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;
```

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.2.

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.3.

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, 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, Adrian Pop, and Martin Sjölund. *Towards Modelica 4 Meta-Programming and Language Modeling with MetaModelica 2.0*. Technical reports in Computer and Information Science, No 10, Linköping University Electronic Press, http://www.ep.liu.se/PubList/ Default.aspx?SeriesID=2550, February 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.

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 200*6 (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. http://en.wikipedia.org/wiki/Partial_evaluation, Last Accessed January 2011.

StephenWolfram. *The Mathematica Book*. 5th Ed. Wolfram Media, Inc, 2003.

Index