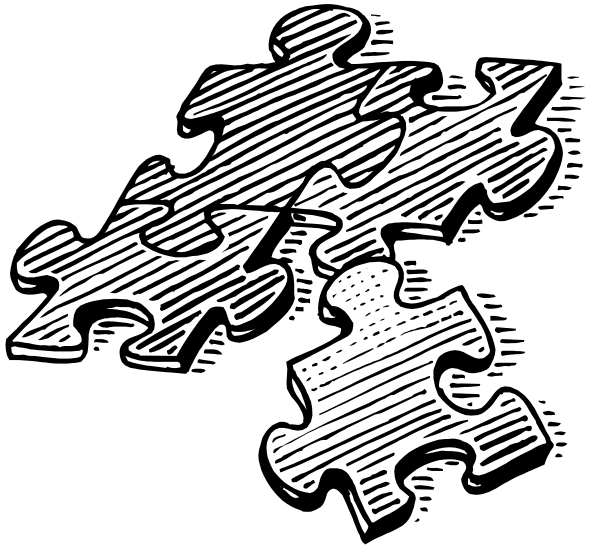
Language Modeling and Symbolic Transformations with Meta-Modelica

Version 0.5, June 2005

Preliminary Incomplete Draft, 2005-06-08



Peter Fritzson

(BRK)

Table of Contents

(BRK)

Preface

012345678**9**012345678**9**012345678**9**012345678**9**012345678**9**012345678**9**012345678**9**012345678**9**01

?? Preface To be written

Linköping, February 2005

Peter Fritzson

Chapter 1

Generation of Language Implementations from Specifications

The implementation of 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.

1.1 Using Meta-Modelica for Programming Language Modeling

The Meta-Modelica specification or 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. 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 and Structured Operational Semantics are also declarative equation-based formalisms. These fit well into the style of the Meta-Modelica 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 Meta-Modelica 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 Meta-Modelica for programming language specification, in a style reminiscent of Natural or 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 Meta-Modelica.

1.2 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. [Pagan81??] which gives an easy to read introduction to several formalisms, including Attribute Grammars and Denotational Semantics. (??Also reference to [Louden2003??] 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 Meta-Modelica formalism fulfils these requirements, and have therefore chosen this formalism for semantics specification in this text.

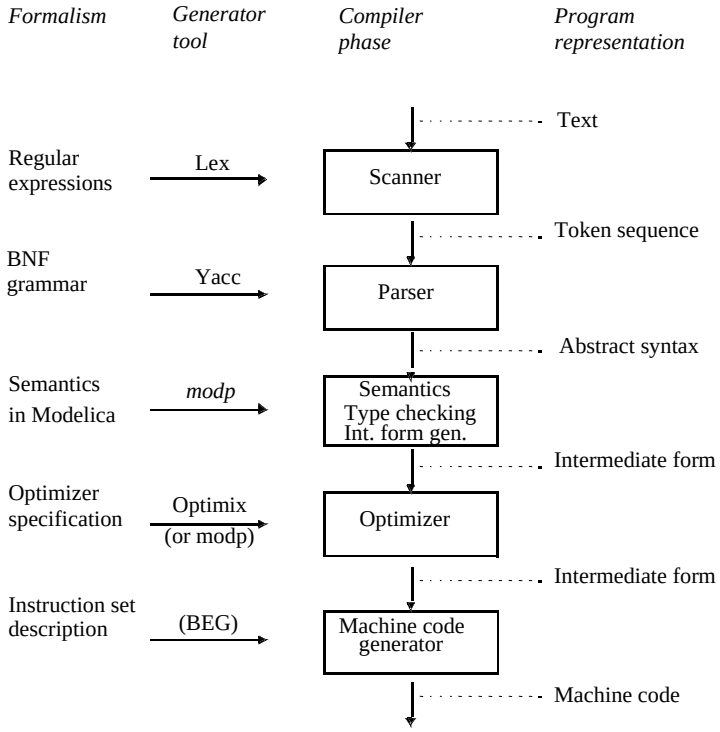


Figure 1-1. Generation of implementations of compiler phases from different formalisms. Meta-Modelica is used to specify the semantics module, which is generated using the tool momc.

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, `momc`, that produces highly efficient implementations in C—roughly of the same efficiency as hand-written ones, and a Modelica debugger for debugging specifications. Modelica 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 [ref??], and BURG [ref??], [refAndersson,Fritzson-95??] have been developed during the late 1980s and early 1990:s. A product version of BEG available in the CoSy compiler generation toolbox [??ref] also includes global register allocation and instruction scheduling. [??also reference the Karlsruhe version]

The optimization phase of compilers is generally hand coded, although some prototypes of optimizer generators have recently appeared. For example, an optimizer generator tool called Optimix [ref??], has appeared as one of the tools in the CoSy [ref??] compiler generation system.

Meta-Modelica 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 Meta-Modelica 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.

1.3 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 [ref??] that sometimes can produce compilers also from specifications of interpreters.

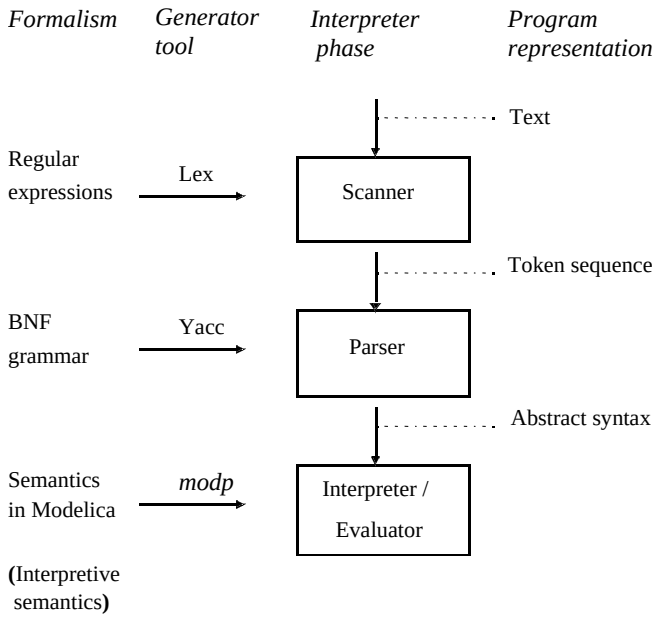


Figure 1-2. Generation of a typical interpreter. The program text is converted into an abstract syntax representation, which is then evaluated by an interpreter generated by the Meta-Modelica momc system. Alternatively, some other intermediate representation such as postfix code can be produced, which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

(BRK)

Chapter 2

Expression Evaluators and Interpreters in Meta-Modelica

We will introduce the topic of language specification in Meta-Modelica through a number of example languages.

The reader who would first prefer a general overview of some language properties of the Meta-Modelica 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 Meta-Modelica 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 Meta-Modelica 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 Meta-Modelica 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 Meta-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;
```

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 Meta-Modelica. Other predefined Meta-Modelica data types are `Real`, `Boolean`, and `String` as well as the parametric type constructors `array`, `list`, and `Option`.

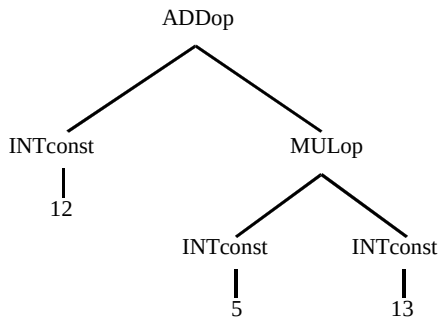


Figure 2-1. Abstract syntax tree of 12+5*13 in the language Exp1.

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 Meta-Modelica 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, also called rules. 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 Meta-Modelica

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

- Match expressions with pattern-matching case rules, 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.

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 `s` with one of the constant strings `"one"`, `"two"` or `"three"`.

```
  String s;
  Real   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:

- Only algebraic equations are allowed as local equations, no differential equations.
- Only locally declared variables (local unknowns) declared by local declarations within the case expression are solved for, or may appear as pattern variables.
- Equations are solved in the order they are declared (this restriction may be removed in the future??).
- There are two variants of these expressions: match-expressions or matchcontinue-expressions. These have identical syntax apart from the keywords `match` or `matchcontinue`.

- If an equation or an expression in a case-branch of a matchcontinue-expression fails, all local variables become unbound, and matching continues with the next branch. However, in case of a match-expression, the whole match-expression will fail if one case-branch fails.

In the following we will primarily use match-expressions 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 `int_add`, `int_sub`, `int_mul`, 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     in_value1;
  output Integer out_value1;
algorithm
 out_value1 :=
  match in_value1
    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 = int_add(v1,v2);  then v3;

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

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

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

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

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

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

This rule 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 rule is to match the argument to `eval` against the special case `INTconst(ival)` of an expression tree. If there is a match, the match variable `ival` 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 rule of `eval`, which is specifying the evaluation of addition nodes labeled `ADDop`:

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

For this rule 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 rule 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 = int_add(v1,v2)` will be checked. This local equation refers to a pre-defined Meta-Modelica function called `int_add` 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 rule.

The rules (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 Meta-Modelica operators such as `int_sub`, `int_mul`, and `int_div`.

The last rule 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 = int_neg(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 `int_add`, `int_sub`, `int_mul`, etc.:

```
function eval
  input  Exp     in_value1;
  output Integer out_value1;
algorithm
 out_value1 :=
  match in_value1
    local Integer v1,v2;
          Exp     e1,e2;
```

```
    case INTconst(v1) then v1;

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

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

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

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

    case NEGop(e1) equation
      v1 = eval(e1);  then -v1;
   end match;
end eval;
```

An even shorter specification can be achieved if all the intermediate variables `v1`, `v2`, etc. are completely eliminated. The temporary variables and the previously shown local equations are internally generated by the Meta-Modelica compiler, and therefore need not be manually specified:

```
function eval
  input  Exp     in_value1;
  output Integer out_value1;
algorithm
 out_value1 :=
  match in_value1
    local Integer v1;  Exp e1,e2;
    case INTconst(v1) then v1;
    case ADDop(e1,e2) then eval(e1) + eval(e2);
    case SUBop(e1,e2) then eval(e1) - eval(e2);
    case MULop(e1,e2) then eval(e1) * eval(e2);
    case DIVop(e1,e2) then eval(e1) / eval(e2);
    case NEGop(e1) then -eval(e1);
  end match;
end eval;
```

In the following we will use verbose or concise specification styles depending on the context.

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 rules 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 x1;  end INTconst;
  record  BINARY    Exp x1;  BinOp op;  Exp x2;  end BINARY;
  record  UNARY     UnOp op;  Exp x1;  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;
```

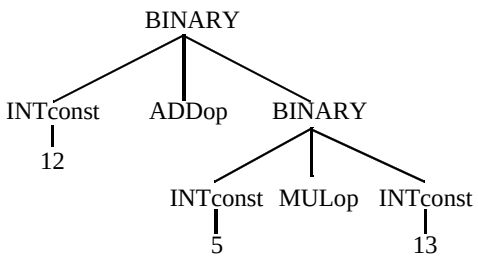


Figure 2-2. A parameterized abstract syntax tree of 12+5*13 in the language Exp1. Compare to the abstract syntax tree in Figure 2-1.

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 x1;  end INT;
  record  BINARY  Exp x1;  BinOp op;  Exp x2;  end BINARY;
  record  UNARY   UnOp op;  Exp x1;  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 rules 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 Meta-Modelica

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 rules for the expression language Exp2, embedded in the functions `eval`, `apply_binop`, and `apply_unop`. As already mentioned, constructor names have been shortened compared to the specification of Exp1. Two rules 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 `apply_binop` and `apply_unop` 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 in_value1;
```

```
  output Integer out_value1;
algorithm
  out_value1:=
   match in_value1
    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 = apply_binop(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 = apply_unop(unop, v1);
      then v2;
  end match;
end eval;
```

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

```
function eval
  input Exp in_value1;
  output Integer out_value1;
algorithm
 out_value1:=
  match in_value1
    local
      Integer ival;  Exp e1,e2,e;
      BinOp binop;  UnOp unop;
    case INT(ival) then ival;
    case BINARY(e1,binop,e2) then apply_binop(binop, eval(e1), eval(e2));
    case UNARY(unop,e) then apply_unop(unop, eval(e));
  end match;
end eval;
```

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

```
function apply_binop
  input BinOp op;
  input Integer arg1;
  input Integer arg2;
```

```
   output Integer out_value1;
 algorithm
  out_value1:=
   match (op,arg1,arg2)
     local Integer v1,v2;
     case (ADD,v1,v2) then v1+v2;
     case (SUB,v1,v2) then v1-v2;
     case (MUL,v1,v2) then v1*v2;
     case (DIV,v1,v2) then v1/v2;
   end match;
 end apply_binop;
```

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: You might wonder why we do not directly reference the function input arguments `arg1` and `arg2` in the case branches, instead of doing a pattern matching to `v1` and `v2`? The reason is a limitation in the current version (April 2005) of the Meta-Modelica subset compiler which prevents you from accessing function input arguments except in match-expression headers.

Finally we present the function `apply_unop` 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 apply_unop
   input UnOp op;
   input Integer arg1;
   output Integer out_value1;
 algorithm
   out_value1:=
    match (op,arg1)
     local Integer v;
     case (NEG,v) then -v;
   end match;
 end apply_unop;
```

For the small language Exp2 the semantic description has become more complicated since we now need three functions, `eval`, `apply_binop` and `apply_unop`, 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 Meta-Modelica

Before continuing the series of language specifications expressed in Meta-Modelica, it is will be useful to say a few words about the Meta-Modelica 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 Meta-Modelica

We have already stated that Meta-Modelica 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 Meta-Modelica function containing match- or matchcontinue-expressions maps inputs to outputs, just as an ordinary function, but also has two additional properties:

- Functions containing match/matchcontinue-expressions can succeed or fail.
- Local backtracking between case rules can occur in matchcontinue-expressions. This means that if a case rule fails because one of its equations or function calls fail, the next rule is tried.

The `fac` example below shows a function calculating factorials. This is an example of using Meta-Modelica not for language specification, but to state a small declarative (i.e., functional) program:

```
function fac
  input Integer in_value1;
  output Integer out_value1;
algorithm
 out_value1:=
  match in_value1
    local  Integer n;
    case 0 then 1;
    case n then if n>0 then n*fac(n-1);
  end match;
end lookup;
```

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

The second rule 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 rule.

2.3.1.1 Handling Failure

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

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

Thus the function will fail, meaning it will not return an ordinary value to the calling function. After a `fail` has occurred in a rule or in some function called from that rule, backtracking takes place, and the next rule in the current function is tried instead.

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

```
function fac_failsafe
  input Integer in_value1;
protected
  Integer dummy_value1;
algorithm
 dummy_value1 :=
  matchcontinue in_value1
    local  Integer n,result;  String str_result;
    case n equation
        str_result = int_string(fac(n));
        print("Res: "); print(str_result);  print("\n");
      then 0;
    case n equation
        failure(result = fac(n));
        print("Cannot apply factorial relation to n."); print("\n");
      then 1;
  end matchcontinue;
end lookup;
```

The function `fac_failsafe` has two rules corresponding to the two cases of correct and incorrect arguments. Since the patterns are overlapping and we need to continue trying the next rule if the first rule fails, we need to use `matchcontinue` instead of `match` which would return immediately with a fail if the first rule fails. We use the **`failure`**(...) primitive to check for failure of the first equation in the second rule.

The first rule 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 Meta-Modelica `print` function.

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

In the second rule 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 rule.

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. There is also a builtin function `bool_success`(*eqarg*) which can be used for testing success or failure in a context where a `Boolean` value is needed. It returns `true` if its equation argument *eqarg* succeeds and `false` if it fails. 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 (see Section **Error! Reference source not found.**) or hash tables. Such representations are commonly used to provide fast lookup in product quality compilers or interpreters.

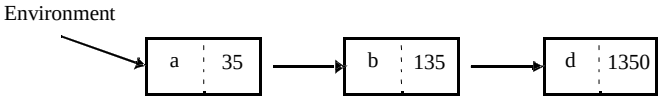


Figure 2-3. An environment represented as a linked list, containing name-value pairs for a, b and d.

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, but will later change implementation (??update?) , see Section **Error! Reference source not found.**, 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 rules have been added compared to the Exp language: one rule for the assignment statement, two rules for the sequence of assignments, one rule for allowing assignments as subexpressions, and finally the program production has been extended to first take a sequence of assignments, then a separating semicolon, and lastly an ending expression.

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

program          : assignments T_SEMIC expression

assignments      :  assignment
                 |  assignments  assignment

assignment       :  ident  T_ASSIGN  expression

expression       :  term
                 |  expression  weak_operator  term

term             :  u_element
                 |  term  strong_operator  u_element

u_element        :  element
                 |  unary_operator  element

element          :  T_INTCONST
                 |  T_LPAREN  expression  T_RPAREN
                 |  T_LPAREN assignment T_RPAREN

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

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

White space represents one or more blanks, tabs or new lines, and is ignored, i.e., no token is returned. A letter is a letter a-z or A-Z or underscore. An identifier (ident) is a letter followed by zero or more letters or digits. A digit is a character within the range 0-9. Digits 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 x1;  end INT;
  record  IDENT   Ident id;  end IDENT;
  record  BINARY  Exp x1;  BinOp op;  Exp x2;  end BINARY;
  record  UNARY   UnOp op;  Exp x1;  end UNARY;
  record  ASSIGN  Ident id;  Exp x1;  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  ListExp x1;  Exp x2;  end PROGRAM;
end Program;

type  ListExp = 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 Meta-Modelica syntax for tuples is: (*item1*, *item2*, ... *itemN*) of which a pair is a special case with two items. The Meta-Modelica `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 x1;  end INT;
  record  IDENT   Ident id;  end IDENT;
  record  BINARY  Exp x1;  BinOp op;  Exp x2;  end BINARY;
  record  UNARY   UnOp op;  Exp x1;  end UNARY;
  record  ASSIGN  Ident id;  Exp x1;  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  ListExp x1;  Exp x2;  end PROGRAM;
end Program;

type ListExp       = 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 rule 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 **Error! Reference source not found.**. If there is no value associated with the identifier, `lookup` will fail.

```
function lookup
  input Env in_env;
  input Ident in_id;
  output Value out_value1;
algorithm
 out_value1:=
  match (in_env,in_id)
    local  Ident id2,id;  Integer value;  Env rest;
    case ( (id2,value) :: rest, id)
      then  if id == id2 then value else lookup(rest,id);
  end match;
end lookup;
```

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

The case rule 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 `in_env`. 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 rule, 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 rules corresponding to the two cases. Since the patterns are overlapping and we need to continue trying the next rule if the first rule fails, we need to use `matchcontinue` instead of `match` which would return immediately with a fail if the first rule fails. We use the **`failure`**(...) primitive to check for failure of the first equation in the second rule.

```
function lookup
  input Env in_env;
  input Ident in_id;
  output Value out_value1;
algorithm
 out_value1:=
  matchcontinue (in_env,in_id)
    local  Ident id2,id;  Integer value;  Env rest;
    /* Identifier id is found in the first pair of the list, and value
     * is returned. */
    case ( (id2,value)::_ ,id)  equation  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(id = id2);  value = lookup(rest, id);
      then value;
  end matchcontinue;
end lookup;
```

The *first* rule, 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 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 rule will be applied.

The *second* rule 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 rule 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(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 rules 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, which is using the builtin primitive `bool_success` to test for success or failure of the equation `value = lookup(env, id)`:

```
function lookupextend
  input Env in_env;
  input Ident in_id;
  output Env out_value1;
  output Value out_value2;
algorithm
 out_value1:=
  match (in_env,in_id)
    local  Env env;  Ident id;  Integer value;
    case (env,id) then
        if bool_success(value = lookup(env, id)) then (env, value);
        else ( (id,0) :: env), 0);
  end match;
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 rules corresponding to the above two rules concerning the occurrence of an identifier. Both rules are using the same pattern `(env,id)`. Here we need to use `matchcontinue` in order to continue matching with the next rules if the current rule fails – a kind of exception handling for fail exceptions. A match-expression would immediately return with a fail if the current rule fails.

```
function lookupextend
  input Env in_value1;
  input Ident in_value2;
  output Env out_value1;
  output Value out_value2;
algorithm
 out_value1:=
  matchcontinue (in_value1,in_value2)
    local  Env env;  Ident id;  Integer 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 rules:

- 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 in_env;
  input Ident in_id;
  input Value in_value3;
  output Env out_value1;
algorithm
 out_value1:=
  match (in_env,in_id,in_value3)
    local  Env env;  Ident id;  Integer value;
    case (env,id,value) then (id,value) :: env;
  end match;
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 rules 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 rule 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 in_value1;
  input Exp in_value2;
  output Env out_value1;
  output Integer out_value2;
algorithm
  out_value1:=
  match (in_value1,in_value2)
    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 rules below specify the evaluation of the binary (`ADD`, `SUB`, `MUL`, `DIV`) and unary (`NEG`) operators. The first rule 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 `apply_binop` 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 rule 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 = apply_binop(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 = apply_unop(unop, v1);
      then (env2,v2);
  end match;
end eval;
```

The functions `apply_binop` and `apply_unop` 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 rules 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.

The language was originally presented by Frank Pagan in his book Formal Specification of Programming Languages [ref??], which gives a very pedagogical introduction to formal specification using several formalisms such as attribute grammars, two-level grammars, operational semantics, denotational semantics and axiomatic semantics. The reader who would like a more in-depth description of PAM and would like to learn about other formalisms is highly recommended to read Pagan's book.

2.5.1 Examples of PAM Programs

A PAM program consists of a series of statements, as in the example below where the factorial of a number `N` is computed. First the number `N` is read from the input stream. Then the special case of factorial of zero is dealt with, giving the value 1. Note that factorial of a negative number is not handled by this program, not even by an error message since there are no strings in this language.

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

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

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

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

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 x1;  end INT;
  record IDENT   Ident id;    end IDENT;
  record BINARY  Exp x1; BinOp op;  Exp x2;  end BINARY;
  record RELATION  Exp x1;  RelOp op;  Exp x3;  end RELATION;
end Exp;

type IdentList = list<Ident>;
uniontype Stmt
  record ASSIGN Ident id; Exp x2; end ASSIGN;       "Id := Exp"
  record IF     Exp x1; Stmt x2; Stmt x3; end IF;   "if Exp then Stmt.."
  record WHILE  Exp x1; Stmt x2;  end WHILE;        " while Exp do Stmt"
  record TODO   Exp x1;  Stmt x2; end TODO;         " to Exp do Stmt..."
  record READ   IdentList x1;  end READ;            "read id1,id2,..."
  record WRITE  IdentList x1;  end WRITE;           "write id1,id2,.."
  record SEQ    Stmt x1;  Stmt x2;  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 x1;  end INTval;
  record BOOLval  Boolean x1; end BOOLval;
end Value;
```

We also introduce a data 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 rules 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 (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 rule specifies evaluation of integer constant leaf nodes (`INT(v)`) which evaluate independently of the environment (because of the 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 Meta-Modelica compiler will have better chances of detecting type errors in the specification.

```
function eval  "Evaluation of expressions in the current environment"
  input Env in_env;
  input Exp in_exp;
  output Value out_value1;
algorithm
  out_value1:=
  match (in_env,in_exp)
    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 rules define the evaluation of identifier leaf nodes (`IDENT(id)`). The first rule describe successful lookup of a variable value in the environment, returning a tagged integer value (`INTval(v)`).

The second rule 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);
```

Alternatively, the evaluation of identifier nodes can be specified by just one rule containing a conditional expression:

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

The last two rules specify evaluation of binary arithmetic operators and boolean relational operators, respectively. These rules first take care of argument evaluation, which thus need not be repeated for each rule in the invoked functions `apply_binop` and `apply_relop` 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 rule 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 = apply_binop(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 = apply_relop(relop, v1, v2); then BOOLval(v3);
  end match;
end eval;
```

2.5.4.2 Arithmetic and Relational Operators

The functions `apply_binop` and `apply_relop` 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 rule to invoke the appropriate predefined Meta-Modelica operation.

```
function apply_binop
            "Apply a binary arithmetic operator to constant integer arguments"
  input BinOp op;
  input Integer arg1;
  input Integer arg2;
  output Integer out_value1;
algorithm
  out_value1:=
```

```
  match (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 match;
end apply_binop;

function apply_relop  "Apply a relation operator, returning a boolean value"
  input RelOp op;
  input Integer arg1;
  input Integer arg2;
  output Boolean out_boolean;
algorithm
  out_boolean :=
  match (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 match;
end apply_relop;
```

2.5.4.3 Statement Evaluation

The `eval_stmt` 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 `eval_stmt` which is `(State, Stmt) => State`. Below we describe the function `eval_stmt` by explaining the semantics of each statement type separately.

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

```
function eval_stmt
    "Statement evaluation: map the current state into a new state"
    input State in_state;
    input Stmt in_stmt;
    output State out_state;
  algorithm
    out_state :=
    match (in_state,in_stmt)
      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`). (??update text)

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

These two rules can be compacted into one rule, using a conditional expression:

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

The next rule defines the semantics of the iterative while-statement. It is fundamentally different from all rules we have previously encountered in that the while construct recursively refers to itself in the local equation of the rule. 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 rule 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 Meta-Modelica compiler can implement this rule 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 rules in the function.

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

The semantics of the while-statement can alternatively be modeled by the two rules below. The first rule, when the comparison evaluates to `false`, returns the current state unchanged. The second rule, 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 = eval_stmt(state,s1);
    state3 = eval_stmt(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 rule, 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 `repeat_eval`.

```
case (state as (env,_,_), TODO(e1,s1))                  // to e1 do s1 ...
  equation
    INTval(n1) = eval(env, e1);
    state2 = repeat_eval(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 rule covers the case of reading into an empty list of variables, which has no effect and returns the current state unchanged. The second rule models actual reading of values from the input stream. First, one item is extracted from the input stream by calling `input_item`, which returns a

modified input stream and a value. The `input_stream` 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) = input_item(istream);
    env2 = update(env, id, INTval(v2));
    state2 = eval_stmt((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 = output_item(ostream,v2);
    state2 = eval_stmt((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 = eval_stmt(state, stmt1);
    state3 = eval_stmt(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 match;

end eval_stmt;
```

2.5.4.4 Auxiliary Functions

The next few subsections defines auxiliary functions, `repeat_eval`, `error`, `input_item`, `output_item`, `lookup`, and `update`, needed by the rest of the PAM specification.

2.5.4.5 Repeated Statement Evaluation

The function `repeat_eval(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 `repeat_eval` for the remaining `n-1` iterations, giving state which is returned.

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

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 `input_item` 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 `input_item` and `output_item` are not updated, they are just dummy streams which give the functions the correct type signatures.

```
function input_item  "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 input_item;
```

The `output_item` 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 output_item  "Write an integer item on the output stream"
  input Stream ostream;
```

```
  input Integer i;
  output Stream ostream2;
 protected
  String s;
algorithm
  s := int_string(i);
  print(s);
  ostream2 := ostream;
end output_item;
```

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 rule 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 rule 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 in_env;
  input Ident in_id;
  output Value out_value;
algorithm
  out_value :=
  match (in_env,in_id)
    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 match;
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 out_env;
algorithm
  out_env := (id,value) :: env;
end update;
```

2.5.4.9 The Complete Interpretive Semantics for PAM

The complete semantics of PAM 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.

```
encapsulated 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 x1;  end INT;
    record IDENT   Ident id;    end IDENT;
    record BINARY  Exp x1; BinOp op;  Exp x2;  end BINARY;
    record RELATION  Exp x1;  RelOp op;  Exp x3;  end RELATION;
  end Exp;

  type IdentList = list<Ident>;
  uniontype Stmt
    record ASSIGN Ident id; Exp x2; end ASSIGN;      "Id := Exp"
    record IF     Exp x1; Stmt x2; Stmt x3; end IF;  "if Exp then Stmt.."
    record WHILE  Exp x1; Stmt x2;  end WHILE;       " while Exp do Stmt"
    record TODO   Exp x1;  Stmt x2; end TODO;        " to Exp do Stmt..."
    record READ   IdentList x1;  end READ;           "read id1,id2,..."
    record WRITE  IdentList x1;  end WRITE;          "write id1,id2,.."
    record SEQ    Stmt x1;  Stmt x2;  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 x1;  end INTval;
    record BOOLval  Boolean x1; end BOOLval;
  end Value;


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

  function eval_stmt
                "Statement evaluation: map the current state into a new state"
    input State in_state;
    input Stmt in_stmt;
    output State out_state;
  algorithm
    out_state :=
    match (in_state,in_stmt)
      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 = eval_stmt(state1, s1);  then state2;
      case (state1 as (env,istream,ostream),IF(comp,s1,s2))     // if false ...
        equation
          BOOLval(false) = eval(env, comp);
          state2 = eval_stmt(state1, s2); then state2;
      case (state,WHILE(comp,s1))                               // while ...
        equation
          state2 = eval_stmt(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 = repeat_eval(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) = input_item(istream);
          env2 = update(env, id, INTval(v2));
          state2 = eval_stmt((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 = output_item(ostream,v2);
          state2 = eval_stmt((env,istream,ostream2), WRITE(rest));  then state2;
      case (state,SEQ(stmt1,stmt2))                              // stmt1 ; stmt2
        equation
          state2 = eval_stmt(state, stmt1);
          state3 = eval_stmt(state2, stmt2); then state3;
      case (state,SKIP()) then state;                        // ; empty statement
    end match;
  end eval_stmt;


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

function eval  "Evaluation of expressions in the current environment"
  input Env in_env;
  input Exp in_exp;
  output Value out_value;
algorithm
  out_value :=
  match (in_env,in_exp)
    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)) then
      if bool_success(v = lookup(env,id)) then v           // Identifier id
      else error("Undefined identifier", id);              // If id not declared,
                          //  give an error message and fail by calling error()
    case (env,BINARY(e1,binop,e2)) equation                // expr1 binop expr2
      INTval(v1) = eval(env, e1);
      INTval(v2) = eval(env, e2);
      v3 = apply_binop(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 = apply_relop(relop, v1, v2); then BOOLval(v3);
  end match;
end eval;
```

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

function apply_binop
            "Apply a binary arithmetic operator to constant integer arguments"
  input BinOp op;
  input Integer arg1;
  input Integer arg2;
  output Integer out_value1;
algorithm
  out_value1 :=
  match (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 match;
end apply_binop;

function apply_relop  "Apply a relation operator, returning a boolean value"
  input RelOp op;
  input Integer arg1;
  input Integer arg2;
  output Boolean out_boolean;
algorithm
  out_boolean :=
  match (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 match;
end apply_relop;



/***************** Auxiliary utility relations *****************/

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 out_value;
algorithm
  out_value :=
  match (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 match;
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 out_env;
algorithm
  out_env := (id,value) :: env;
end update;

function repeat_eval  "repeatedly evaluate stmt n times"
  input State state;
  input Integer n;
  input Stmt stmt;
  output State out_state;
algorithm
  out_state :=
  if n <= 0 then state                                    /* n <= 0 */
  else repeat_eval(eval_stmt(state, stmt), n-1, stmt);  /* eval n times */
end repeat_eval;

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 input_item  "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 input_item;

function output_item  "Write an integer item on the output stream"
  input Stream ostream;
  input Integer i;
  output Stream ostream2;
 protected
  String s;
algorithm
  s := int_string(i);
```

```
  print(s);
  ostream2 := ostream;
end output_item;
```

2.6 AssignTwoType – Introducing Typing

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

2.6.1 Concrete Syntax of AssignTwoType

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

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

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

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

The lexical specification follows below. One new token type, `T_REALCONST`, has been introduced compared to the Assignments language. The regular expression rcon1 represents a real constant that must contain a dot, whereas `rcon2` must contain an exponent. Any real constant must contain either a dot or an exponent. The ? 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 rules 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  ExpList x1;  Exp x2;  end PROGRAM;
end Program;

uniontype Exp
  record INT    Integer x1;  end INT;
  record REAL   Real x1;  end REAL;
  record BINARY Exp x1;  BinOp op;  Exp x2;  end BINARY;
  record UNARY  UnOp op;  Exp x1;  end UNARY;
  record ASSIGN Ident id;  Exp x1;  end ASSIGN;
  record IDENT  Ident id;  end IDENT;
end Exp;

type ExpList = 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 x1;  end INTval;
  record REALval  Real x1;  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 x1;  Integer x2;  end INT2;
  record REAL2  Real x1;  Real x2;  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 rules 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 rules, we introduce a type lattice (see Section 0), and use the function `type_lub` (for least upper bound of types; Section 0) which derives the resulting type and inserts possibly needed type conversions. This reduces the number of needed rules 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 rules.

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 rule for `Real` constants, and collected all binary operators together into two rules, and unary operators into two additional rules. The rules for assignments and variable identifiers are the same as before.

We show the application of some rules 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 rule 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, `apply_real_binop` 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 in_exp in current environment in_env, returning
  a possibly updated environment out_env, and an out_value 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 in_env;
  input Exp in_exp;
  output Env out_env;
  output Value out_value;
algorithm
  (out_env,out_value):=
  matchcontinue (in_env,in_exp)
    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 integer_binop integer"
      equation
        (env1,v1) = eval(env, e1);
        (env2,v2) = eval(env, e2);
        INT2(x,y) = type_lub(v1, v2);
        z = apply_int_binop(binop, x, y); then (env2,INTval(z));
    case (env,BINARY(e1,binop,e2))      "integer/real real_binop 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 = apply_real_binop(binop, x, y); then (env2,REALval(z));
    case (env,UNARY(unop,e))            "integer_unop  exp"
      equation
        (env1,INTval(x)) = eval(env, e);
        y = apply_int_unop(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 = apply_real_unop(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 match;
end eval;
```

2.6.3.2 Type Lattice and Least Upper Bound

One general way to partially avoid the potential "combinatorial explosion" of semantic rules 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.

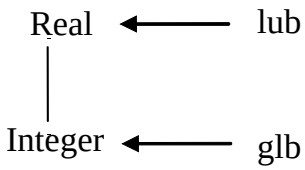


Figure 2-4. Simple type lattice for types integer and real. The least upper bound (lub) is real; the greatest lower bound (glb) is integer.

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

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

These rules 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 rules 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 in_value1;
  input Value in_value2;
  output Ty2 out_ty2;
algorithm
  out_ty2:=
```

```
  match (in_value1,in_value2)
    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 match;
end type_lub;
```

2.6.3.3 Binary and Unary Operators

The essential properties of binary arithmetic operators are described below in the functions `apply_int_binop` and `apply_real_binop`, respectively. Argument evaluation has been taken care of by the two rules for binary operators in the function `eval`, and thus need not be repeated for each rule. 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 (`apply_int_binop`) or `Real` values (`apply_real_binop`).

```
function apply_int_binop  "Apply integer binary operator"
  input BinOp in_binop1;
  input Integer in_integer2;
  input Integer in_integer3;
  output Integer out_integer;
algorithm
  out_integer:=
  match (in_binop1,in_integer2,in_integer3)
    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 match;
end apply_int_binop;

function apply_real_binop  "Apply real binary operator"
  input BinOp in_binop1;
  input Real in_real2;
  input Real in_real3;
  output Real out_real;
algorithm
  out_real:=
  match (in_binop1,in_real2,in_real3)
    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 match;
end apply_real_binop;
```

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

```
function apply_int_unop  "Apply integer unary operator"
  input UnOp in_unop;
  input Integer in_integer;
  output Integer out_integer;
algorithm
  out_integer:=
  match (in_unop,in_integer)
    local Integer x;
    case (NEG(),x) then -x;
  end match;
end apply_int_unop;

function apply_real_unop  "Apply real unary operator"
  input UnOp in_unop;
  input Real in_real;
  output Real out_real;
algorithm
  out_real:=
  match (in_unop,in_real)
    local Real x;
    case (NEG(),x) then -.x;
  end match;
end apply_real_unop;
```

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 out_value;
algorithm
  out_value :=
  match (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 match;
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 in_env;
  input Ident in_ident;
  output Env out_env;
  output Value out_value;
algorithm
  (out_env,out_value):=
  matchcontinue (in_env,in_ident)
    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 match;
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 out_env;
algorithm
  out_env := (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 module 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 below, using five modules: `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 modules in order to reference items declared in those modules. References to names defined in other modules must be prefixed by the defining module name followed by a dot, as in `Absyn.ASSIGN()` when referencing the `ASSIGN` constructor from module `Absyn`.

2.7.1 The Main Module (?? update)

The main module implements the prompt-read-eval-print loop as the function `evalprog`, which accepts the initial environment `initial` containing only true and false exported from module `Eval`, and loops indefinitely??.

The main module of the PamDecl evaluator calls `ScanParse` to read and parse text from the standard input, and `Eval` to evaluate and print the results. (?? update ??)

```
package Main
  import PamDecl.ScanParse;
  import PamDecl.Eval;

type StringList = list<String>;

function mainprogram
  input StringList;
  output Boolean dummy;
algorithm
  ast := ScanParse.scanparse();
  ast := Eval.evalprog(ast);
  dummy := true; //?? should really call mainprogram recursively to have a loop
??
end mainprogram;

end Main;
```

2.7.2 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 PamDecl.Absyn;

function scanparse
  output Absyn.Prog ast;
external "C";

end ScanParse;
```

2.7.3 Absyn

Add more explanations??

```
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 x1;  end INTCONST;
  record REALCONST  Real x1;  end REALCONST;
  record BINARY  Expr x1;  BinOp x2;  Expr x3;  end BINARY;
  record UNARY  UnOp x1;  Expr x2;  end UNARY;
  record RELATION  Expr x1;  RelOp x2;  Expr x3;  end RELATION;
  record VARIABLE  Ident x1;  end VARIABLE;
end Expr;

type StmtList = list<Stmt>;

uniontype Stmt
  record ASSIGN  Ident x1;  Expr x2;  end ASSIGN;
  record WRITE  Expr x1;  end WRITE;
  record NOOP  end NOOP;
  record IF  Expr x1;  StmtList x2;  StmtList x3;  end IF;
  record WHILE  Expr x1;  StmtList x2;  end WHILE;
end Stmt;

type StmtList = list<Stmt>;

uniontype Decl
  record NAMEDECL  Ident x1;  Ident x2;  end NAMEDECL;
end Decl;

type DeclList = list<Decl>;
```

```
uniontype Prog
  record PROG  DeclList x1;  StmtList x2;  end PROG;
end Prog;

end Absyn;
```

2.7.4 Env

Add more explanations??

```
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 x1;  end INTVAL;
  record REALVAL  Real x1;  end REALVAL;
  record BOOLVAL  Boolean x1;  end BOOLVAL;
end Value;

uniontype Value2  "Values for real-integer type lattice conversions"
  record INTVAL2  Integer x1;  Integer x2;  end INTVAL2;
  record REALVAL2  Real x1;  Real x2;  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 in_env;
  input Ident in_ident;
  output Value out_value;
algorithm
  out_value:=
  match (in_env,in_ident)
```

```
    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 match;
end lookup;


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

  input Env in_env;
  input Ident in_ident;
  output Type out_type;
algorithm
  out_type:=
  match (in_env,in_ident)
    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 match;
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 out_env;
algorithm
  out_env := BIND((id,ty,value) :: env)
end update;

end Env;
```

2.7.5 Eval

Add more explanations ??

```
package Eval

import PamDecl.Absyn;
import PamDecl.Env;

function evalprog  "Evaluating a program means to evaluate the list of
   statements with an initial environment containing just standard definitions."
```

```
  input Absyn.Prog in_prog;
  output Boolean dummy;
algorithm
  dummy:=
  match (in_prog)
    local
      type Env_BindList = list<Env.Bind>;
      Env_BindList env1;
    case Absyn.PROG(decls,stmts)
      equation
        env1 = Env.initial;
        env2 = eval_decl_list(env1, decls);
        env3 = eval_stmt_list(env2, stmts); then true;
  end match;
end evalprog;


/* Evaluation of statements */

function eval_stmt "Evaluate a single statement. Pass environment forward."
  input Env.Env in_env;
  input Absyn.Stmt in_stmt;
  output Env.Env out_env;
algorithm
  out_env:=
  matchcontinue (in_env,in_stmt)
    local
      type Env_BindList = list<Env.Bind>;
      Env.Value v;
      Env.Type ty;
      Env_BindList env,env1;
      String id;
      Absyn.Expr e;
    case (env,Absyn.ASSIGN(id,e))
      equation
        v = eval_expr(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 = eval_expr(env, e);
        print("Error: assignment mismatch or variable missing\n"); then fail();
    case (env,Absyn.WRITE(e))
      equation
        v = eval_expr(env, e);
        print_value(v); then env;
    case (env,Absyn.NOOP()) then env;
    case (env,Absyn.IF(e,c,_))
      equation
        Env.BOOLVAL(true) = eval_expr(env, e);
        env1 = eval_stmt_list(env, c); then env1;
    case (env,Absyn.IF(e,_,a))
      equation
```

```
        Env.BOOLVAL(false) = eval_expr(env, e);
        env1 = eval_stmt_list(env, a); then env1;
    case (env,Absyn.WHILE(e,ss))
      equation
        Env.BOOLVAL(true) = eval_expr(env, e);
        env1 = eval_stmt_list(env, ss);
        env2 = eval_stmt(env1, Absyn.WHILE((e,ss))); then env2;
    case (env,Absyn.WHILE(e,ss))
      equation
        Env.BOOLVAL(false) = eval_expr(env, e); then env;
    case (env,Absyn.IF(e,_,a))
      equation
        Env.BOOLVAL(false) = eval_expr(env, e);
        env1 = eval_stmt_list(env, a); then env1;
    case (env,Absyn.WHILE(e,ss))
      equation
        Env.BOOLVAL(true) = eval_expr(env, e);
        env1 = eval_stmt_list(env, ss);
        env2 = eval_stmt(env1, Absyn.WHILE((e,ss))); then env2;
    case (env,Absyn.WHILE(e,ss))
      equation
        Env.BOOLVAL(false) = eval_expr(env, e); then env;
  end match;
end eval_stmt;

function eval_stmt_list "Evaluate a list of statements in an environment.
                         Pass environment forward"
  input Env.Env in_env;
  input Absyn.StmtList in_stmtlist;
  output Env.Env out_env;
algorithm
  out_env:=
  match (in_env,in_stmtlist)
    local
      type Env_BindList = list<Env.Bind>;
      Env_BindList env;
    case (env, {}) then env;
    case (env, s :: ss)
      equation
        env1 = eval_stmt(env, s);
        env2 = eval_stmt_list(env1, ss); then env2;
  end match;
end eval_stmt_list;

/* Evaluation of Declarations */

function eval_decl "Evaluate a single declaration. Pass environment forward."
  input Env.Env in_env;
  input Absyn.Decl in_decl;
  output Env.Env out_env;
algorithm
  out_env:=
  match (in_env,in_decl)
    local
```

```
      type Env_BindList = list<Env.Bind>;
      Env_BindList 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 match;
end eval_decl;

function eval_decl_list
                "Evaluate a list of declarations, extending the environent."
  input Env.Env in_env;
  input Absyn.DeclList in_decllist;
  output Env.Env out_env;
algorithm
  out_env:=
  match (in_env,in_decllist)
    local
      type Env_BindList = list<Env.Bind>;
      Env_BindList env;
    case (env,nil) then env;
    case (env,s :: ss)
      equation
        env1 = eval_decl(env, s);
        env2 = eval_decl_list(env1, ss); then env2;
  end match;
end eval_decl_list;


function eval_expr "Evaluate a single expression in an environment. Return
                    the new value. Expressions do not change environments. "
  input Env.Env in_env;
  input Absyn.Expr in_expr;
  output Env.Value out_value;
algorithm
  out_value:=
  matchcontinue (in_env,in_expr)
    local
      type Env_BindList = list<Env.Bind>;
      Env_BindList env;
      Env.Value v1,v2;
      Real c1,c2,v3;
      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)) then Env.REALVAL(v);
case (env,Absyn.BINARY(e1,binop,e2))                "Binary operators"
  equation
    v1 = eval_expr(env, e1);
    v2 = eval_expr(env, e2);
    Env.INTVAL2(c1,c2) = binary_lub(v1, v2);
    v3 = apply_int_binary(binop, c1, c2); then Env.INTVAL(v3);
case (env,Absyn.BINARY(e1,binop,e2))
  equation
    v1 = eval_expr(env, e1);
    v2 = eval_expr(env, e2);
    Env.REALVAL2(c1,c2) = binary_lub(v1, v2);
    v3 = apply_real_binary(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 Real v1,v2;
  equation
    Env.INTVAL(v1) = eval_expr(env, e1);
    v2 = apply_int_unary(unop, v1); then Env.INTVAL(v2);
case (env,Absyn.UNARY(unop,e1))
  equation
    Env.REALVAL(v1) = eval_expr(env, e1);
    v2 = apply_real_unary(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))              "relational operators"
    local Boolean v3;
  equation
    v1 = eval_expr(env, e1);
    v2 = eval_expr(env, e2);
    Env.INTVAL2(c1,c2) = binary_lub(v1, v2);
    v3 = apply_int_relation(relop, c1, c2); then Env.BOOLVAL(v3);
case (env,Absyn.RELATION(e1,relop,e2))
  equation
    v1 = eval_expr(env, e1);
    v2 = eval_expr(env, e2);
    Env.REALVAL2(c1,c2) = binary_lub(v1, v2);
    v3 = apply_real_relation(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 identifier lookup"
  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 eval_expr;



function binary_lub  "Type lattice; int --> real"
  input Env.Value in_value1;
  input Env.Value in_value2;
  output Env.Value2 out_value2;
algorithm
  out_value2:=
  match (in_value1,in_value2)
    local Real v1,v2;
    case (Env.INTVAL(v1),Env.INTVAL(v2)) then Env.INTVAL2((v1,v2));
    case (Env.REALVAL(v1),Env.REALVAL(v2))
        local Integer v1; then Env.REALVAL2((v1,v2));
    case (Env.INTVAL(v1),Env.REALVAL(v2))
        local Integer v2;
      equation
        c1 = int_real(v1); then Env.REALVAL2((c1,v2));
    case (Env.REALVAL(v1),Env.INTVAL(v2))
      equation
        c2 = int_real(v2); then Env.REALVAL2((v1,c2));
  end match;
end binary_lub;

function promote "Promotion and type check "
  input Env.Value in_value;
  input Env.Type in_type;
  output Env.Value out_value;
algorithm
  out_value:=
  match (in_value,in_type)
    local Integer v;
    case (Env.INTVAL(v),Env.INTTYPE()) then Env.INTVAL(v);
    case (Env.REALVAL(v),Env.REALTYPE()) then Env.REALVAL(v);
    case (Env.BOOLVAL(v),Env.BOOLTYPE()) then Env.BOOLVAL(v);
    case (Env.INTVAL(v),Env.REALTYPE())
      equation
        v2 = int_real(v); then Env.REALVAL(v2);
  end match;
end promote;

/*  Auxiliary functions for applying the binary operators */

function apply_int_binary  "Apply integer binary operators"
  input Absyn.BinOp in_binop1;
  input Integer in_integer2;
  input Integer in_integer3;
  output Integer out_integer;
algorithm
  out_integer:=
```

```
  match (in_binop1,in_integer2,in_integer3)
    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 match;
end apply_int_binary;

function apply_real_binary  "Apply real binary operators"
  input Absyn.BinOp in_binop1;
  input Real in_real2;
  input Real in_real3;
  output Real out_real;
algorithm
  out_real:=
  match (in_binop1,in_real2,in_real3)
    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 match;
end apply_real_binary;


/* Auxiliary functions for applying the unary operators */

function apply_int_unary  "Apply integer unary operators"
  input Absyn.UnOp in_unop;
  input Integer in_integer;
  output Integer out_integer;
algorithm
  out_integer:=
  match (in_unop,in_integer)
    local Real v1;
    case (Absyn.NEG(),v1) then -v1;
  end match;
end apply_int_unary;

function apply_real_unary  "Apply unary real operators"
  input Absyn.UnOp in_unop;
  input Real in_real;
  output Real out_real;
algorithm
  out_real:=
  match (in_unop,in_real)
    local Integer v1;
    case (Absyn.NEG(),v1) then -. v1;
  end match;
end apply_real_unary;

/* Auxiliary functions for applying the relational operators  */
```

```
function apply_int_relation  "Apply integer relational operators"
  input Absyn.RelOp in_relop1;
  input Integer in_integer2;
  input Integer in_integer3;
  output Boolean out_boolean;
algorithm
  out_boolean:=
  match (in_relop1,in_integer2,in_integer3)
    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 match;
end apply_int_relation;

function apply_real_relation  "Apply real relational operators"
  input Absyn.RelOp in_relop1;
  input Real in_real2;
  input Real in_real3;
  output Boolean out_boolean;
algorithm
  out_boolean:=
  match (in_relop1,in_real2,in_real3)
    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 match;
end apply_real_relation;

function print_value  "Evaluate the 'write' statement, i.e., print a value"
  input Env.Value in_value;
  output Boolean dummy;
algorithm
  dummy:=
  match (in_value)
    local
      String vstr;
      Real v;
    case Env.INTVAL(v)
      equation
        vstr = int_string(v);
        print(vstr);
        print("\n"); then true;
    case Env.REALVAL(v)
      equation
        vstr = real_string(v);
        print(vstr);
```

```
        print("\n"); then true;
    case Env.BOOLVAL(true)
      equation
        print("true\n"); then true;
    case Env.BOOLVAL(false)
      equation
        print("false\n"); then true;
  end match;
end print_value;


end Eval;
```

2.8 Summary

In this chapter we present a series of small example languages to introduce Meta-Modelica 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 rules, 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.

(BRK)

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.

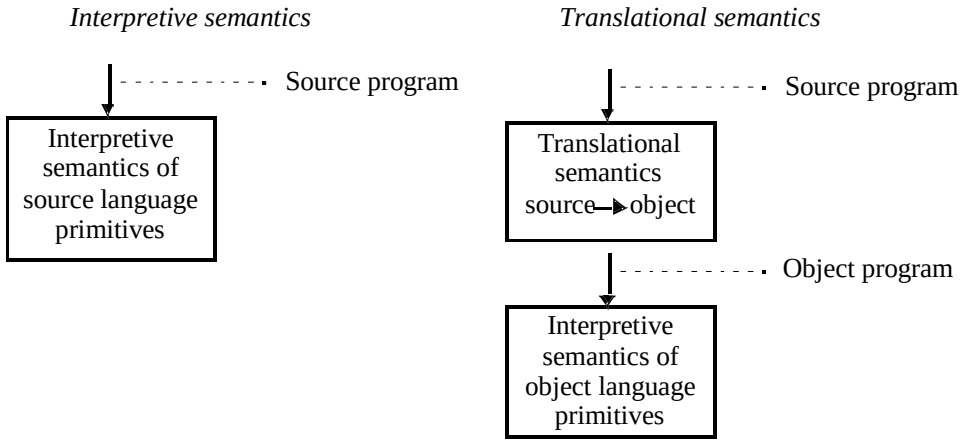


Figure 3-1. A comparison between an interpretive semantics and translational semantics. In an interpretive semantics, the computational meaning of source language primitives are directly defined, e.g. using Meta-Modelica. In a translational semantics, the meaning is defined as a translation to object 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 Meta-Modelica 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 1.2.

There are also techniques based on partial evaluation [refs??see big book from 92 or 94], for the generation of compilers from certain styles of interpretive semantics. However, these techniques often give unpredictable results and performance problems. Therefore, in the rest of this text we will exclusively use translational semantics as a basis for practical compiler generation.

In fact, writing translational semantics is usually not harder than writing interpretive semantics. One just has to keep in mind that the semantics is described in two parts: the meaning of source language primitives in terms of (a translation to) target language primitives, and the meaning of the target primitives themselves. A simplified picture of compiler generation from translational semantics is shown in Figure 3-2.

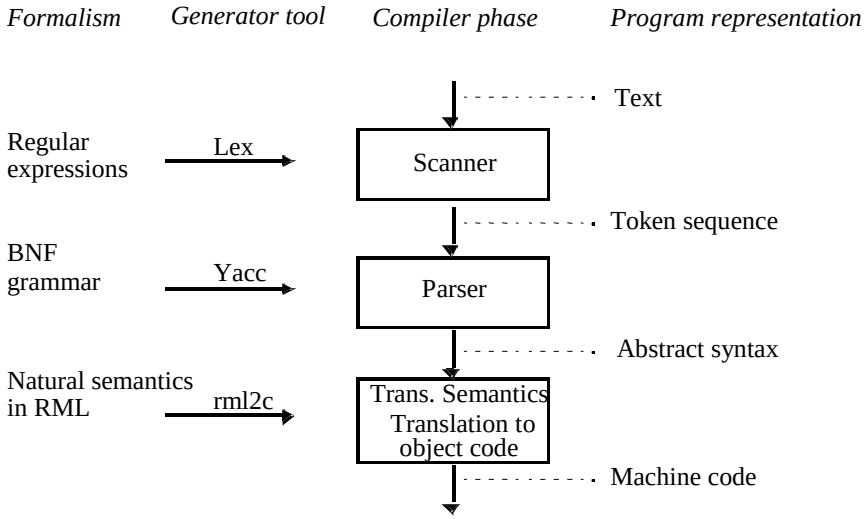


Figure 3-2. Simplified version of compiler generation based on translational semantics. The semantics of a language is specified directly in terms of object code primitives. In comparison to Figure 1-1, the optimization and final code generation phases have been excluded.

3.1 Translating PAM to Machine Code

As an introduction translational semantics, we will specify the translational semantics of a simple language, with the goal of generating a compiler from this language to machine code. The simple PAM language has already been described, and an interpretive semantics has been given in Section 2.5. This makes it a natural first choice for a translational semantics. In Chapter 3 of [ref Pagan], an attribute grammar style translational semantics of PAM can be found. It is instructive to compare the attribute grammar specification to the Meta-Modelica 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 [ref Pagan] 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 Meta-Modelica. 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 Meta-Modelica 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 x1;  end I;
  record N  Integer x1;  end N;
  record T  Integer x1;  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 x1;  Moperand x2;  end MB;    /* Binary arith ops */
  record MJ     MCondJmp x1; MLab x2;  end MJ;       /* Conditional jumps */
  record MJMP   Mlab x1;     end MJMP;
  record MLOAD  MIdTemp x1;  end MLOAD;
  record MSTO   MIdTemp x1;  end MSTO;
  record MGET   MIdent x1;  end MGET;
  record MPUT   MIdent x1;  end MPUT;
  record MLABEL MLab x1;  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 x1;  end INT;
  record IDENT   Ident id;    end IDENT;
  record BINARY  Exp x1; BinOp op;  Exp x2;  end BINARY;
  record RELATION  Exp x1;  RelOp op;  Exp x3;  end RELATION;
end Exp;

type IdentList = list<Ident>;
uniontype Stmt
  record ASSIGN Ident id; Exp x2; end ASSIGN;       "Id := Exp"
  record IF     Exp x1; Stmt x2; Stmt x3; end IF;   "if Exp then Stmt.."
  record WHILE  Exp x1; Stmt x2;  end WHILE;        " while Exp do Stmt"
  record TODO   Exp x1;  Stmt x2; end TODO;         " to Exp do Stmt..."
  record READ   IdentList x1;  end READ;            "read id1,id2,..."
  record WRITE  IdentList x1;  end WRITE;           "write id1,id2,.."
  record SEQ    Stmt x1;  Stmt x2;  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 Meta-Modelica 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 `trans_expr relation` together with two small help functions `trans_binop` and `gentemp`. The `trans_binop` 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 `trans_expr` function contains essentially all semantics of PAM arithmetic expressions. The first two axioms 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 `trans_expr` function below:

```
function trans_expr "Arithmetic expression translation"
  type MCode_MCodeList = list<MCode.Mcode>;
  input Absyn.Exp in_exp;
  output MCode_MCodeList out_MCode_MCodeList;
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 rule 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 rule). 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 rule follows below.

```
case Absyn.BINARY(e1,binop,e2) "Arith binop: simple case, expr2 is just an
                                identifier or constant:   expr1 binop expr2"
  equation
    cod1 = trans_expr(e1);
    list(MCode.MLOAD(operand2)) = trans_expr(e2);  "Condition expr2 simple";
```

```
    opcode = trans_binop(binop);
    cod3 = list_append(cod1, list(MCode.MB(opcode,operand2))); then cod3;
```

The second rule 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 rule 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 `list_append6` appends the elements of six argument lists, whereas the standard `list_append` 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 = trans_expr(e1);
    cod2 = trans_expr(e2);
    opcode = trans_binop(binop);
    t1 = gentemp();
    t2 = gentemp();
    cod3 = list_append6(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 rules for binary arithmetic operations overlap. The first rule 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 rule which covers both the simple and the general case. From a semantic point of view,

the first rule is not needed since the second rule specifies the same semantics for simple arithmetic expressions as the second rule, even though the second rule will give rise to more instructions in the translated code. Still, it is not incorrect to keep the first rule, since the PAM semantics is not changed by it.

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

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

```
function trans_expr  "Arithmetic expression translation"
  type MCode_MCodeList = list<MCode.Mcode>;
  input Absyn.Exp in_exp;
  output MCode_MCodeList out_MCode_MCodeList;
algorithm
  out_MCode_MCodeList:=
  match (in_exp)
    local
      Integer v;
      String id;
      MCode_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 = trans_expr(e1);
        list(MCode.MLOAD(operand2)) = trans_expr(e2);
        opcode = trans_binop(binop)   " expr2 simple ";
        cod3 = list_append(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 = trans_expr(e1);
        cod2 = trans_expr(e2);
        opcode = trans_binop(binop);
        t1 = gentemp();
        t2 = gentemp();
        cod3 = list_append6(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 match;
end trans_expr;

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

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

function list_append6
  replaceable type Type_a;
  type Type_aList = list<Type_a>;
  input Type_aList l1;
  input Type_aList l2;
  input Type_aList l3;
  input Type_aList l4;
  input Type_aList l5;
  input Type_aList l6;
  output Type_aList l16;
protected
  Type_aList l13,l46;
algorithm
  l13 = list_append3(l1, l2, l3);
  l46 = list_append3(l4, l5, l6);
  l16 = list_append(l13, l46);
end list_append6;
```

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 `trans_relop`, 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 trans_relop  "Translate relation operator"
  input Absyn.RelOp in_relop;
  output MCode.MCondJmp out_mcondjmp;
algorithm
  out_mcondjmp:=
  match (in_relop)
    case Absyn.EQ() then MCode.MJNP();  " Jump on Negative or Positive "
    case Absyn.LE() then MCode.MJP();   " Jump on Positive "
    case Absyn.LT() then MCode.MJPZ();  " Jump on Positive or Zero "
    case Absyn.GT() then MCode.MJNZ();  " Jump on Negative or Zero "
    case Absyn.GE() then MCode.MJN();   " Jump on Negative "
    case Absyn.NE() then MCode.MJZ();   " Jump on Zero "
  end match;
end trans_relop;
```

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

```
function trans_comparison  "Translate comparison relation operator"
  type MCode_MCodeList = list<MCode.Mcode>;
  input Absyn.Comparison in_comparison;
  input MCode.MLab in_mlab;
  output MCode_MCodeList out_MCode_MCodeList;
```

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 rule in the `trans_comparison` function translates according to this pattern, as shown below. The first rule 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 = trans_expr(e1);
    list(MCode.MLOAD(operand2)) = trans_expr(e2);
    jmpop = trans_relop(relop);
    cod3 = list_append3(cod1, {MCode.MB(MCode.MSUB(),operand2)},
                              {MCode.MJ(jmpop,lab)} );  then cod3;
```

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

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

function trans_comparison   "translation of a comparison: expr1 relop expr2
    Example call:  trans_comparison(RELATION(INDENT(x), GT, INT(5)), L(10))"
  type MCode_MCodeList = list<MCode.Mcode>;
  input Absyn.Comparison in_comparison;
  input MCode.MLab in_mlab;
  output MCode_MCodeList out_MCode_MCodeList;
algorithm
  out_MCode_MCodeList :=
  matchcontinue (in_comparison,in_mlab)
    local
      MCode_MCodeList 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 rule), when expr2 is  a simple
    * identifier or constant:
    *   code for expr1
    *   SUB operand2
    *   conditional jump to lab
    *
    * or a general code pattern (second rule), which is needed when expr2
    * is more complicated than a simple identifier or constant:
    *   code for expr1
    *   STO temp1
    *   code for expr2
    *   SUB temp1
    *   conditional jump to lab
    */
    case (Absyn.RELATION(e1,relop,e2),lab)   "Simple case, expr1 relop expr2"
      equation
        cod1 = trans_expr(e1);
        list(MCode.MLOAD(operand2)) = trans_expr(e2);
        jmpop = trans_relop(relop);
        cod3 = list_append3(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 = trans_expr(e1);
        cod2 = trans_expr(e2);
        jmpop = trans_relop(relop);
        t1 = gentemp();
        cod3 = list_append5(cod1, {MCode.MSTO(t1)}, cod2,
          {MCode.MB(MCode.MSUB(),t1)}, {MCode.MJ(jmpop,lab)} );
          then cod3;
  end matchcontinue;
end trans_comparison;
```

```
function trans_relop  "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 in_relop;
  output MCode.MCondJmp out_mcondjmp;
algorithm
  out_mcondjmp:=
  match (in_relop)
    case Absyn.EQ() then MCode.MJNP();  " Jump on Negative or Positive "
    case Absyn.LE() then MCode.MJP();   " Jump on Positive "
    case Absyn.LT() then MCode.MJPZ();  " Jump on Positive or Zero "
    case Absyn.GT() then MCode.MJNZ();  " Jump on Negative or Zero "
    case Absyn.GE() then MCode.MJN();   " Jump on Negative "
    case Absyn.NE() then MCode.MJZ();   " Jump on Zero "
  end match;
end trans_relop;
```

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 rules of the function `trans_stmt`.

The first rule 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 rule:

```
case Absyn.ASSIGN(id,e1)      /* Assignment Statement translation:
                                 map the current state into a new state */
  equation
    cod1 = trans_expr(e1);
    cod2 = list_append(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 axiom below:

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

Translation of if-statements is more complicated. There are two rules, 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 rule:

```
case Absyn.IF(comp,s1,Absyn.SKIP)                 /* if comp then s1 */
```

```
    equation
      s1cod = trans_stmt(s1);
      l1 = genlabel();
      compcod = trans_comparison(comp, l1);
      cod3 = list_append3(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 rule:

```
case Absyn.IF(comp,s1,s2)              /* if comp then s1 else s2 */
  equation
    s1cod = trans_stmt(s1);
    s2cod = trans_stmt(s2);
    l1 = genlabel();
    l2 = genlabel();
    compcod = trans_comparison(comp, l1);
    cod3 = list_append6(
       compcod, s1cod,
       {MCode.MJMP(l2)},
       {MCode.MLABEL(l1)},
       s2cod,
       {MCode.MLABEL(l2)} );  then cod3;
```

This second rule 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 rule.

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 rule:

```
case Absyn.WHILE(comp,s1)           " while ... "
  equation
    bodycod = trans_stmt(s1);
    l1 = genlabel();
    l2 = genlabel();
    compcod = trans_comparison(comp, l2);
    cod3 = list_append5(
```

```
        {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 rule:

```
case Absyn.TODO(e1,s1)              " to e1 do s1 ... "
  equation
    tocod = trans_expr(e1);
    bodycod = trans_stmt(s1);
    t1 = gentemp();
    l1 = genlabel();
    l2 = genlabel();
    cod3 = list_append10(
      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 axiom and rule, stating that reading an empty list of variables produces an empty sequence of GET instructions, whereas the rule specifies emission of one GET instruction for the first identifier in the non-empty list, and then recursively invokes trans_stmt for the rest of the identifiers in the list. The axiom and the rule follows below:

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

case Absyn.READ(id :: idlist_rest)       " read id1,id2,... "
  equation
    cod2 = trans_stmt(Absyn.READ(idlist_rest));
      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 axiom and rule:

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

case Absyn.WRITE(id :: idlist_rest)         " write id1,id2,... "
  equation
    cod2 = trans_stmt(Absyn.WRITE(idlist_rest));
      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 rule below:

```
case Absyn.SEQ(stmt1,stmt2)                 " stmt1 ; stmt2 "
  equation
    cod1 = trans_stmt(stmt1);
    cod2 = trans_stmt(stmt2);
    cod3 = list_append(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 trans_program below:

```
function trans_program   "Translate a whole program"
  type MCode_MCodeList = list<MCode.Mcode>;
  input Absyn.Stmt progbody;
  output MCode_MCodeList programcode;
protected
  MCode_MCodeList cod1;
algorithm
  cod1 := trans_stmt(progbody);
  programcode := list_append(cod1, {MCode.MHALT()});
end trans_program;
```

Finally, the complete translational semantics of PAM statements is presented below as the rules and axioms of the function `trans_stmt`.

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

function trans_stmt         "Statement translation"
  type MCode_MCodeList = list<MCode.Mcode>;
  input Absyn.Stmt in_stmt;
  output MCode_MCodeList out_MCode_MCodeList;
algorithm
  out_MCode_MCodeList:=
  match (in_stmt)
    local
      type StringList = list<String>;
      MCode_MCodeList 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;
      StringList idlist_rest;

    case Absyn.ASSIGN(id,e1)     /* Assignment Statement translation:
                                    map the current state into a new state */
      equation
        cod1 = trans_expr(e1);
        cod2 = list_append(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 = trans_stmt(s1);
        l1 = genlabel();
        compcod = trans_comparison(comp, l1);
        cod3 = list_append3(compcod, s1cod, {MCode.MLABEL(l1)} ); then cod3;

    case Absyn.IF(comp,s1,s2)             /* if comp then s1 else s2 */
      equation
        s1cod = trans_stmt(s1);
        s2cod = trans_stmt(s2);
        l1 = genlabel();
        l2 = genlabel();
        compcod = trans_comparison(comp, l1);
        cod3 = list_append6(
           compcod, s1cod,
           {MCode.MJMP(l2)},
           {MCode.MLABEL(l1)},
           s2cod,
           {MCode.MLABEL(l2)} );  then cod3;

    case Absyn.WHILE(comp,s1)          " while ... "
      equation
        bodycod = trans_stmt(s1);
        l1 = genlabel();
        l2 = genlabel();
```

```
        compcod = trans_comparison(comp, l2);
        cod3 = list_append5(
          {MCode.MLABEL(l1)},
          compcod, bodycod,
          {MCode.MJMP(l1)},
          {MCode.MLABEL(l2)} ); then cod3;

    case Absyn.TODO(e1,s1)              " to e1 do s1 ... "
      equation
        tocod = trans_expr(e1);
        bodycod = trans_stmt(s1);
        t1 = gentemp();
        l1 = genlabel();
        l2 = genlabel();
        cod3 = list_append10(
          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 :: idlist_rest)      " read id1,id2,... "
      equation
        cod2 = trans_stmt(Absyn.READ(idlist_rest));
          then MCode.MGET(MCode.I(id) :: cod2);

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

    case Absyn.WRITE(id :: idlist_rest)         " write id1,id2,... "

      equation
        cod2 = trans_stmt(Absyn.WRITE(idlist_rest));
          then MCode.MPUT(MCode.I(id) :: cod2);

    case Absyn.SEQ(stmt1,stmt2)                 " stmt1 ; stmt2 "
      equation
        cod1 = trans_stmt(stmt1);
        cod2 = trans_stmt(stmt2);
  end match;

end trans_stmt;
```

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 `emit_assembly` function and associated functions below. This is not really part of the translational semantics. Here, Meta-Modelica is used as a semi-functional programming language, to implement the desired conversion. The `print` primitive has been included in the standard Meta-Modelica 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 emit_assembly  "Print an MCode instruction"
  input MCodeList in_mcodelist;
  output Boolean dummy;
protected
  type MCodeList = list<MCode.Mcode>;
algorithm
  dummy:=
  match (in_mcodelist)
    local
      MCode.Mcode instr;
      MCodeList rest;
    case ({}) then true;
    case (instr :: rest)
      equation
        emit_instr(instr);
        emit_assembly(rest); then true;
  end match;
end emit_assembly;

function emit_instr
  input MCode.Mcode in_MCode;
  output Boolean dummy;
algorithm
  dummy:=
  match (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 = mbinop_to_str(mbinop);
        emit_op_operand(op, mopr); then true;
    case (MCode.MJ(jmpop,mlab))
      equation
        op = mjmpop_to_str(jmpop);
        emit_op_operand(op, mlab); then true;
    case (MCode.MJMP(mlab))
      equation
        emit_op_operand("J", mlab); then true;
    case (MCode.MLOAD(mopr))
```

```
      equation
        emit_op_operand("LOAD", mopr); then true;
    case (MCode.MSTO(mopr))
      equation
        emit_op_operand("STO", mopr); then true;
    case (MCode.MGET(mopr))
      equation
        emit_op_operand("GET", mopr); then true;
    case (MCode.MPUT(mopr))
      equation
        emit_op_operand("PUT", mopr); then true;
    case (MCode.MLABEL(mlab))
      equation
        emit_moperand(mlab);
        print("\tLAB\n"); then true;
    case (MCode.MHALT())
      equation
        print("\tHALT\n"); then true;
  end match;
end emit_instr;

function emit_op_operand
  input String opstr;
  input MCode.MOperand mopr;
algorithm
  print("\t");
  print(opstr);
  print("\t");
  emit_moperand(mopr);
  print("\n");
end emit_op_operand;

function emit_int
  input Integer i;
protected
  String s;
algorithm
  s := int_string(i);
  print(s);
end emit_int;

function emit_moperand
  input MCode.MOperand in_moperand;
  output Boolean dummy;
algorithm
  dummy:=
  match (in_moperand)
    local
      String id;
      Integer number,labno,tempnr;
    case (MCode.I(id))
      equation
        print(id); then true;
    case (MCode.N(number))
```

```
      equation
        emit_int(number); then true;
    case (MCode.L(labno))
      equation
        print("L");
        emit_int(labno); then true;
    case (MCode.T(tempnr))
      equation
        print("T");
        emit_int(tempnr); then true;
  end match;
end emit_moperand;

function mbinop_to_str
  input MCode.MBinOp in_mbinop;
  output String out_string;
algorithm
  out_string:=
  match (in_mbinop)
    case (MCode.MADD()) then "ADD";
    case (MCode.MSUB()) then "SUB";
    case (MCode.MMULT()) then "MULT";
    case (MCode.MDIV()) then "DIV";
  end match;
end mbinop_to_str;

function mjmpop_to_str
  input MCode.MCondJmp in_mcondjmp;
  output String out_string;
algorithm
  out_string:=
  match (in_mcondjmp)
    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 match;
end mjmpop_to_str;

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.trans_program(program);
  Emit.emit_assembly(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 Meta-Modelica.

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.

(?? a good idea to define such an abstract machine here, in the style of a small steps semantics).

3.3 Building and Running the PAM Translator

3.3.1 Building the PAM Translator

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`.

The files can be copied from (??update) `/home/pelab/pub/pkg/rml/current/bookexamples/examples/pamtrans`.

The executable is built by typing:

```
sen20%12 make pamtrans
```

3.3.2 Source Files for PAM Translator

3.3.2.1 lexer.l

```
%{
#include "gram.h"
#include "yacclib.h"
#include "rml.h"
#include "absyn.h"

typedef void *rml_t;
extern rml_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=(rml_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 = (rml_t) mk_scon(s);
        return T_IDENT;
}
gram.y
%{
#include <stdio.h>
#include "yacclib.h"
#include "rml.h"
#include "absyn.h"

typedef void *rml_t;
#define YYSTYPE rml_t

extern rml_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) {

}
```

3.3.2.2 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 x1;  end INT;
  record IDENT   Ident id;    end IDENT;
  record BINARY  Exp x1; BinOp op;  Exp x2;  end BINARY;
  record RELATION  Exp x1;  RelOp op;  Exp x3;  end RELATION;
end Exp;

type IdentList = list<Ident>;
uniontype Stmt
  record ASSIGN Ident id; Exp x2; end ASSIGN;      "Id := Exp"
  record IF     Exp x1; Stmt x2; Stmt x3; end IF;  "if Exp then Stmt.."
  record WHILE  Exp x1; Stmt x2;  end WHILE;       " while Exp do Stmt"
  record TODO   Exp x1;  Stmt x2; end TODO;        " to Exp do Stmt..."
  record READ   IdentList x1;  end READ;           "read id1,id2,..."
  record WRITE  IdentList x1;  end WRITE;          "write id1,id2,.."
```

```
  record SEQ    Stmt x1;  Stmt x2;  end SEQ;       "Stmt1; Stmt2"
  record SKIP  end SKIP;                           " ; empty stmt"
end Stmt;

end Absyn;
```

3.3.2.3 Trans.mo

```
package Trans

import Absyn;
import MCode;


function trans_program   "Translate a whole program"
  type MCode_MCodeList = list<MCode.Mcode>;
  input Absyn.Stmt progbody;
  output MCode_MCodeList programcode;
protected
  MCode_MCodeList cod1;
algorithm
  cod1 := trans_stmt(progbody);
  programcode := list_append(cod1, {MCode.MHALT()});
end trans_program;


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

function trans_stmt         "Statement translation"
  type MCode_MCodeList = list<MCode.Mcode>;
  input Absyn.Stmt in_stmt;
  output MCode_MCodeList out_MCode_MCodeList;
algorithm
  out_MCode_MCodeList:=
  match (in_stmt)
    local
      type StringList = list<String>;
      MCode_MCodeList 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;
      StringList idlist_rest;

    case Absyn.ASSIGN(id,e1)     /* Assignment Statement translation:
                                    map the current state into a new state */
      equation
        cod1 = trans_expr(e1);
        cod2 = list_append(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 = trans_stmt(s1);
        l1 = genlabel();
```

```
    compcod = trans_comparison(comp, l1);
    cod3 = list_append3(compcod, s1cod, {MCode.MLABEL(l1)} ); then cod3;

case Absyn.IF(comp,s1,s2)              /* if comp then s1 else s2 */
  equation
    s1cod = trans_stmt(s1);
    s2cod = trans_stmt(s2);
    l1 = genlabel();
    l2 = genlabel();
    compcod = trans_comparison(comp, l1);
    cod3 = list_append6(
       compcod, s1cod,
       {MCode.MJMP(l2)},
       {MCode.MLABEL(l1)},
       s2cod,
       {MCode.MLABEL(l2)} );  then cod3;

case Absyn.WHILE(comp,s1)           " while ... "
  equation
    bodycod = trans_stmt(s1);
    l1 = genlabel();
    l2 = genlabel();
    compcod = trans_comparison(comp, l2);
    cod3 = list_append5(
      {MCode.MLABEL(l1)},
      compcod, bodycod,
      {MCode.MJMP(l1)},
      {MCode.MLABEL(l2)} ); then cod3;

case Absyn.TODO(e1,s1)              " to e1 do s1 ... "
  equation
    tocod = trans_expr(e1);
    bodycod = trans_stmt(s1);
    t1 = gentemp();
    l1 = genlabel();
    l2 = genlabel();
    cod3 = list_append10(
      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 :: idlist_rest)      " read id1,id2,... "
  equation
    cod2 = trans_stmt(Absyn.READ(idlist_rest));
      then MCode.MGET(MCode.I(id) :: cod2);
```

```
    case Absyn.WRITE({})then {};                " write {} "
    case Absyn.WRITE(id :: idlist_rest)         " write id1,id2,... "

      equation
        cod2 = trans_stmt(Absyn.WRITE(idlist_rest));
          then MCode.MPUT(MCode.I(id) :: cod2);
    case Absyn.SEQ(stmt1,stmt2)                 " stmt1 ; stmt2 "
      equation
        cod1 = trans_stmt(stmt1);
        cod2 = trans_stmt(stmt2);
  end match;

end trans_stmt;


function trans_expr  "Arithmetic expression translation"
  type MCode_MCodeList = list<MCode.Mcode>;
  input Absyn.Exp in_exp;
  output MCode_MCodeList out_MCode_MCodeList;
algorithm
  out_MCode_MCodeList:=
  match (in_exp)
    local
      Integer v;
      String id;
      MCode_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 = trans_expr(e1);
        list(MCode.MLOAD(operand2)) = trans_expr(e2);
        opcode = trans_binop(binop)   " expr2 simple ";
        cod3 = list_append(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 = trans_expr(e1);
        cod2 = trans_expr(e2);
        opcode = trans_binop(binop);
        t1 = gentemp();
        t2 = gentemp();
        cod3 = list_append6(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 match;
end trans_expr;

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

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

function list_append3
  replaceable type Type_a;
  type Type_aList = list<Type_a>;
  input Type_aList l1;
  input Type_aList l2;
  input Type_aList l3;
  output Type_aList l13;
protected
  Type_aList l12;
algorithm
  l12 := list_append(l1, l2);
  l13 := list_append(l12, l3);
end list_append3;

function list_append5
  replaceable type Type_a;
  type Type_aList = list<Type_a>;
  input Type_aList l1;
  input Type_aList l2;
  input Type_aList l3;
  input Type_aList l4;
  input Type_aList l5;
  output Type_aList l15;
protected
```

```
  Type_aList l13;
algorithm
  l13 := list_append3(l1, l2, l3);
  l15 := list_append3(l13, l4, l5);
end list_append5;

function list_append6
  output Boolean dummy;
algorithm
  dummy:=
  match (true)
    local
      replaceable type Type_a;
      type Type_aList = list<Type_a>;
      Type_aList l13,l46,l16,l1,l2,l3,l4,l5,l6;
    case (l1,l2,l3,l4,l5,l6)
      equation
        l13 = list_append3(l1, l2, l3);
        l46 = list_append3(l4, l5, l6);
        l16 = list_append(l13, l46); then l16;
  end match;
end list_append6;

function list_append10
  replaceable type Type_a;
  type Type_aList = list<Type_a>;
  input Type_aList l1;
  input Type_aList l2;
  input Type_aList l3;
  input Type_aList l4;
  input Type_aList l5;
  input Type_aList l6;
  input Type_aList l7;
  input Type_aList l8;
  input Type_aList l9;
  input Type_aList l10;
  output Type_aList l110;
protected
  Type_aList l15;
algorithm
  l15 := list_append5(l1, l2, l3, l4, l5);
  l110 := list_append6(l15, l6, l7, l8, l9, l10);
end list_append10;


relation trans_binop:  Absyn.BinOp => MCode.MBinOp  =
  axiom  trans_binop(Absyn.ADD) =>  MCode.MADD
  axiom  trans_binop(Absyn.SUB)  =>  MCode.MSUB
  axiom  trans_binop(Absyn.MUL)  =>  MCode.MMULT
  axiom  trans_binop(Absyn.DIV)  =>  MCode.MDIV
end

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

```
function trans_comparison   "translation of a comparison: expr1 relop expr2
    Example call:  trans_comparison(RELATION(INDENT(x), GT, INT(5)), L(10))"
  type MCode_MCodeList = list<MCode.Mcode>;
  input Absyn.Comparison in_comparison;
  input MCode.MLab in_mlab;
  output MCode_MCodeList out_MCode_MCodeList;
algorithm
  out_MCode_MCodeList :=
  matchcontinue (in_comparison,in_mlab)
    local
      MCode_MCodeList 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 rule), when expr2 is  a simple
    * identifier or constant:
    *   code for expr1
    *   SUB operand2
    *   conditional jump to lab
    *
    * or a general code pattern (second rule), which is needed when expr2
    * is more complicated than a simple identifier or constant:
    *   code for expr1
    *   STO temp1
    *   code for expr2
    *   SUB temp1
    *   conditional jump to lab
    */
    case (Absyn.RELATION(e1,relop,e2),lab)   "Simple case, expr1 relop expr2"
      equation
        cod1 = trans_expr(e1);
        list(MCode.MLOAD(operand2)) = trans_expr(e2);
        jmpop = trans_relop(relop);
        cod3 = list_append3(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 = trans_expr(e1);
        cod2 = trans_expr(e2);
        jmpop = trans_relop(relop);
        t1 = gentemp();
        cod3 = list_append5(cod1, {MCode.MSTO(t1)}, cod2,
          {MCode.MB(MCode.MSUB(),t1)}, {MCode.MJ(jmpop,lab)} );
          then cod3;
  end matchcontinue;
end trans_comparison;


function trans_relop  "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 in_relop;
  output MCode.MCondJmp out_mcondjmp;
algorithm
  out_mcondjmp:=
  match (in_relop)
    case Absyn.EQ() then MCode.MJNP();  " Jump on Negative or Positive "
    case Absyn.LE() then MCode.MJP();   " Jump on Positive "
    case Absyn.LT() then MCode.MJPZ();  " Jump on Positive or Zero "
    case Absyn.GT() then MCode.MJNZ();  " Jump on Negative or Zero "
    case Absyn.GE() then MCode.MJN();   " Jump on Negative "
    case Absyn.NE() then MCode.MJZ();   " Jump on Zero "
  end match;

end trans_relop;
```

3.3.2.4 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 x1;  end I;
  record N  Integer x1;  end N;
  record T  Integer x1;  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 x1;  Moperand x2;  end MB;    /* Binary arith ops */
  record MJ     MCondJmp x1; MLab x2;  end MJ;       /* Conditional jumps */
  record MJMP   Mlab x1;     end MJMP;
  record MLOAD  MIdTemp x1;  end MLOAD;
  record MSTO   MIdTemp x1;  end MSTO;
```

```
  record MGET   MIdent x1;  end MGET;
  record MPUT   MIdent x1;  end MPUT;
  record MLABEL MLab x1;  end MLABEL;
  record MHALT  end MHALT;
end MCode;

end MCode;
```

3.3.2.5 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 emit_assembly  "Print an MCode instruction"
  input MCodeList in_mcodelist;
  output Boolean dummy;
  type MCodeList = list<MCode.Mcode>;
algorithm
  dummy:=
  match (in_mcodelist)
    local
      MCode.Mcode instr;
      MCodeList rest;
    case ({}) then true;
    case (instr :: rest)
      equation
        emit_instr(instr);
        emit_assembly(rest); then true;
  end match;
end emit_assembly;

function emit_instr
  input MCode.Mcode in_MCode;
  output Boolean dummy;
algorithm
  dummy:=
  match (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 = mbinop_to_str(mbinop);
        emit_op_operand(op, mopr); then true;
    case (MCode.MJ(jmpop,mlab))
      equation
        op = mjmpop_to_str(jmpop);
        emit_op_operand(op, mlab); then true;
```

```
    case (MCode.MJMP(mlab))
      equation
        emit_op_operand("J", mlab); then true;
    case (MCode.MLOAD(mopr))
      equation
        emit_op_operand("LOAD", mopr); then true;
    case (MCode.MSTO(mopr))
      equation
        emit_op_operand("STO", mopr); then true;
    case (MCode.MGET(mopr))
      equation
        emit_op_operand("GET", mopr); then true;
    case (MCode.MPUT(mopr))
      equation
        emit_op_operand("PUT", mopr); then true;
    case (MCode.MLABEL(mlab))
      equation
        emit_moperand(mlab);
        print("\tLAB\n"); then true;
    case (MCode.MHALT())
      equation
        print("\tHALT\n"); then true;
  end match;
end emit_instr;

function emit_op_operand
  input String opstr;
  input MCode.MOperand mopr;
algorithm
  print("\t");
  print(opstr);
  print("\t");
  emit_moperand(mopr);
  print("\n");
end emit_op_operand;

function emit_int
  input Integer i;
protected
  String s;
algorithm
  s := int_string(i);
  print(s);
end emit_int;

function emit_moperand
  input MCode.MOperand in_moperand;
  output Boolean dummy;
algorithm
  dummy:=
  match (in_moperand)
    local
      String id;
      Integer number,labno,tempnr;
```

```
    case (MCode.I(id))
      equation
        print(id); then true;
    case (MCode.N(number))
      equation
        emit_int(number); then true;
    case (MCode.L(labno))
      equation
        print("L");
        emit_int(labno); then true;
    case (MCode.T(tempnr))
      equation
        print("T");
        emit_int(tempnr); then true;
  end match;
end emit_moperand;

function mbinop_to_str
  input MCode.MBinOp in_mbinop;
  output String out_string;
algorithm
  out_string:=
  match (in_mbinop)
    case (MCode.MADD()) then "ADD";
    case (MCode.MSUB()) then "SUB";
    case (MCode.MMULT()) then "MULT";
    case (MCode.MDIV()) then "DIV";
  end match;
end mbinop_to_str;

function mjmpop_to_str
  input MCode.MCondJmp in_mcondjmp;
  output String out_string;
algorithm
  out_string:=
  match (in_mcondjmp)
    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 match;
end mjmpop_to_str;

end Emit;
```

3.3.2.6 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
  type MCodeList = list<MCode.Mcode>;
  Absyn.Stmt program;
  MCodeList mcode;
algorithm
  program := Parse.parse();
  mcode := Trans.trans_program(program);
  Emit.emit_assembly(mcode);
end main;

end Main;
```

3.3.2.7 Parse.mo

```
package Parse
  import Absyn;

function parse
  output Absyn.Stmt out_stmt;

  external "C" ;
end parse;

end Parse;
```

3.3.2.8 parse.c

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include "rml.h"

#ifndef RML_INSPECTBOX
#define RML_INSPECTBOX(d,h,p)
(RML_ISIMM((d)=(p))?0:(((h)=(void*/RML_GETHDR((p))),0))

#define rml_prim_deref_imm(x) x
#endif

void Parse_5finit(void) {}

void *absyntree;

RML_BEGIN_LABEL(Parse__parse) {
    void *a0, *a0hdr;
    RML_INSPECTBOX(a0, a0hdr, rmlA0);
    if( a0hdr == RML_IMMEDIATE(RML_UNBOUNDHDR) )
        RML_TAILCALLK(rmlFC);
    else {
        if(yyparse()==0) {
```

```
            rmlA0 = absyntree;
            RML_TAILCALLK(rmlSC);
        }
        else RML_TAILCALLK(rmlFC);
    }
}
```

makefile

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

# VARIABLES

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

# 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.rml
      $(MOMC) -c main.rml

# 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.rml
	$(MOMC) -c absyn.rml

# TRANSLATION

trans.o:  trans.c
trans.c trans.h:  trans.rml absyn.h
	$(MOMC) -c trans.rml

# EMISSION

emit.o:  emit.c
emit.c emit.h:  emit.rml
	$(MOMC) -c emit.rml

# INTERMEDIATE FORM

mcode.o:  mcode.c
mcode.c mcode.h:  mcode.rml
	$(MOMC) -c mcode.rml


# 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>
```

3.4 Translational Semantics for Symbolic Differentiation

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

```
uniontype Exp
  record RCONST Real x1; end RCONST;
  record PLUS  Exp x1; Exp x2; end PLUS;
  record SUB   Exp x1; Exp x2; end SUB;
  record MUL   Exp x1; Exp x2; end MUL;
  record DIV   Exp x1; Exp x2; end DIV;
  record NEG   Exp x1;         end NEG;
  record IDENT String name; end IDENT;
  record CALL  Exp id; list<Exp> args;  end CALL;
```

```
  record AND   Exp x1; Exp x2; end AND;
  record OR    Exp x1; Exp x2; end OR;
  record LESS  Exp x1; Exp x2; end LESS;
  record GREATER Exp x1; Exp x2;  end GREATER;
end Exp;
```

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

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

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

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

```
function difft "Symbolic differentiation of expression with respect to time"
  input  Exp expr;
  input  list<IDENT> timevars;
  output Exp diffexpr;
algorithm
 diffexpr :=
  match (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 list_member(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 match;

 end difft;
```

3.5 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 Meta-Modelica rules.

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.

(BRK)

Chapter 4

Getting Started – Practical Details (Needs Update)

This chapter provides information about a number of technical details that the reader will need to know in order to get started using the Meta-Modelica momc generator system. This includes information about where the momc system resides, how to invoke the `momc` program generator, how to compile and link generated code, how to run the Meta-Modelica 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 Meta-Modelica system a few changes in the environment need to be done. Note that these changes are non portable and will only work at the Department of Computer and Information Science at Linköping University, Sweden.

In order to get the correct settings for the Meta-Modelica environment one need to add some modules.

```
module initadd labs/pelab pelab-before pelab-pub-before rml   (?? Sun Solaris
only)
```

The module labs/pelab sets up the module path. In order to run an emacs that supports the Modelica-mode ??? is added. The module momc??? sets up the Modelica environment. Two environment variables are set by the rml module: the variable ???`RMLHOME`, which is set to the directory where the complete system of Meta-Modelica resides and `RMLRUNTIME` which is set to the directory of the Meta-Modelica runtime files (bin, lib and include) for sparc solaris2 is located.

To set import the Meta-Modelica emacs mode rml-mode write the following as the first thing in your .emacs file: (?? Sun Solaris??)

```
(setq load-path (cons
             (expand-file-name (concat (getenv "RMLHOME") "/elisp"))
             load-path))
```

The tools lex and yacc can be found in `/user/ccs/bin`, but if the paths have been set up correctly one need not worry about this.

The reader may copy the example files from the `/home/pelab/pub/pkg/rml/current/` bookexamples directory or type them in from the examples in this chapter. Preferrably copy the whole directory with the command:

```
cp -r /home/pelab/pub/pkg/rml/current/bookexamples/ ./myrmlexamples
```

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 `calc` 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)`. 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. (?? this description is only valid for a Unix or Linux shell??)

```
sen20%10 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 calc
[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 Meta-Modelica, 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.rml` contains an interpretive style Meta-Modelica specification and abstract syntax of the Exp1 language in Meta-Modelica form, here within the single Meta-Modelica 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.

4.2.2.2 Generated Source Files

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

- The files `exp1.c` and `exp1.h` are generated by the `momc` translator. 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 also needed. (?? Unix only??)

- The files `yacclib.c` and `yacclib.h` contain some basic primitive routines needed in the course of building abstract syntax tree nodes during parsing. Most of these routines are not called directly by the user. Instead they are typically invoked via the tree building macros defined in `exp1.h`. Some routines (e.g. `mk_icon`, `mk_rcon`, `mk_scon`, `mk_nil`) for building Modelica-type integer, real and string constants (and nil), are also defined in yacclib.c.

- The file `rml.h` contains definitions and macros for calling the Meta-Modelica runtime system and predefined functions (located in `$RMLRUNTIME/include/plain`).
- The file `librml.a` is a library of all Meta-Modelica runtime system routines and predefined functions (located in `$RMLRUNTIME/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 C compiler (gcc) is used here. Library files and header files are found in `$RMLRUNTIME/{include,lib}` if not available in the current directory. The usual make dependencies are specified. The command:

```
make calc
```

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

```
make clean
```

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

```
# Makefile for building the Exp1 calculator
#
# ??Note: LDFLAGS, CFLAGS are non-portable for some Unix systems

# VARIABLES

SHELL = /bin/sh
LDLIBS = -ll -lrml
LDFLAGS = -L$(RMLRUNTIME)/lib/plain/
CC = gcc
CFLAGS = -I$(RMLRUNTIME)/include/plain/ -g



# EVERYTHING
all:  calc


# MAIN PROGRAM

CALCOBJS= main.o lexer.o parser.o yacclib.o exp1.o
calc: $(CALCOBJS)
      $(CC) $(LDFLAGS) $(CALCOBJS) $(LDLIBS) -o calc

main.o:       main.c exp1.h

# LEXER

lexer.o:  lexer.c parser.h exp1.h
lexer.c:  lexer.l
      lex -t lexer.l >lexer.c

# PARSER
```

```
parser.o:  parser.c exp1.h
parser.c parser.h:  parser.y
      yacc -d parser.y
      mv y.tab.c parser.c
      mv y.tab.h parser.h


# ABSTRACT SYNTAX and EVALUATION

exp1.o:  exp1.c
exp1.c exp1.h:     exp1.rml
      momc -c exp1.rml

# AUX

clean:
      -rm calc $(CALCOBJS) lexer.c parser.c parser.h exp1.c exp1.h
```

4.2.3 Source Files for the Exp1 Calculator

Below we present the three source files `lexer.l`, `parser.y`, and `exp1.rml`, 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 momc when processing the abstract syntax definitions in `exp1.rml`.

The routine `mk_icon` (from `yacclib.h`) builds Meta-Modelica compatible integer constants that can be passed to Meta-Modelica constructors such as `exp1.INTconst`, here callable as `exp1__INTconst`.

```
/* file lexer.l */
%{
#include "parser.h"
#include "yacclib.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}+         ;

%%

rml_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 momc from the definition of the Exp1 abstract syntax in the module `exp1` that can be found in the file `exp1.rml`. 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 "yacclib.h"
#include "rml.h"
#include "exp1.h"

typedef void *rml_t;
#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.rml

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 Meta-Modelica package `Exp1`. For larger specifications it is customary to place the definition of abstract syntax in a module 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     in_value1;
  output Integer out_value1;
algorithm
 out_value1 :=
  match in_value1
    local Integer v1,v2;
          Exp     e1,e2;
    case INTconst(v1) then v1;

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

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

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

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

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

end eval;
```

4.2.3.4 main.c

See Section 4.2.4 for more information.

4.2.4 Calling Meta-Modelica from C — main.c (?? To be updated)

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

The main program ties the different modules together and initializes the Meta-Modelica 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 Meta-Modelica runtime system. In fact, for each module `M` written in Meta-Modelica, 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 Meta-Modelica functions are still 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 Meta-Modelica tagged integer representation to the ordinary C integer representation before being printed. This conversion is handled by `RML_UNTAGFIXNUM`.

The special Meta-Modelica 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 `rml.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"

typedef void * rml_t;
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 modules */

  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 `scanner.c`, `parser.h`, `parser.c`, `exp1.h`, and `exp1.c` in Section 4.2.2.2. The Meta-Modelica 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` Meta-Modelica functions, which is rather unreadable C code that is not so interesting to look at.

Additionally, we describe the header file `yacclib.h` of the library file `yacclib.c`, which contains low level routines necessary 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 module. 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 Yacclib.h

The header file `yacclib.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 Meta-Modelica 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 Meta-Modelica `Option` type which is convenient for representing optional syntactic constructs.

Finally, the routines `mk_box0` to `mk_box5` construct abstract syntax tree nodes of arity 0 to 5. 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`.

```
/* yacclib.h */

extern int yylineno;                    /* generated by lex */

extern char *yytok2str(int token);      /* uses yytoks[] from yacc + -DYYDEBUG *
/

extern void error(const char *fmt, ...);

extern void *alloc_bytes(unsigned nbytes);
extern void *alloc_words(unsigned nwords);

extern void print_icon(FILE*, void*/;
extern void print_rcon(FILE*, void*/;
extern void print_scon(FILE*, void*/;

extern void *mk_icon(int);

extern void *mk_rcon(double);
extern void *mk_scon(char*/;
extern void *mk_nil(void);
extern void *mk_cons(void*, void*/;
extern void *mk_none(void);
extern void *mk_some(void*/;
extern void *mk_box0(unsigned ctor);
extern void *mk_box1(unsigned ctor, void*/;
extern void *mk_box2(unsigned ctor, void*, void*/;
extern void *mk_box3(unsigned ctor, void*, void*, void*/;
extern void *mk_box4(unsigned ctor, void*, void*, void*, void*/;
extern void *mk_box5(unsigned ctor, void*, void*, void*, void*, void*/;
```

4.3 An Evaluator for PAMDECL

4.3.1 Running the PAMDECL Evaluator

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

```
sen20%10 cat|pamdecl

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: (??only for Unix)

```
sen20%11 pamdecl > 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.rml` (page 55), `env.rml` (page 55), `eval.rml` (page 56), `lexer.l`, `parser.y`, `main.rml`, `scanparse.rml`, `scanparse.c`, `yacclib.c`, `yacclib.h` and `makefile`.

The files can be copied from `/home/pelab/pub/pkg/rml/current/bookexamples/examples/pamdecl` (??update location) or typed from the above pages and Section 4.3.3 below.

The executable is built by typing:

```
sen20%12 make pamdecl
```

4.3.3 Source Files for PAMDECL Evaluator

For `Absyn.mo`, `Env.mo`, and `Eval.mo` see Section 2.7.

4.3.3.1 lexer.l

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

#include "absyn.h"

typedef void *rml_t;
extern rml_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=(rml_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=(rml_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 = (rml_t) mk_scon(s);
  return T_IDENT;
}
```

4.3.3.2 parser.y

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

typedef void *rml_t;
#define YYSTYPE rml_t
extern rml_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 stmt_list 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);}

stmt_list
        :
            { $$ = mk_nil();}
        | stmt stmt_list
            { $$ = mk_cons($1,$2); }

stmt
        : simple_stmt T_SEMICOLON
        | combined_stmt

simple_stmt
        : assign_stmt
        | write_stmt
        | noop_stmt


combined_stmt
        : if_stmt
        | while_stmt

assign_stmt
: T_IDENT T_ASSIGN expr
            { $$ = Absyn__ASSIGN($1,$3);}

write_stmt
        : T_WRITE expr
            { $$ = Absyn__WRITE($2);}

noop_stmt
        :
            { $$ = Absyn__NOOP;}

if_stmt
```

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

while_stmt
        : T_WHILE expr T_DO stmt_list T_END T_WHILE
            { $$ = Absyn__WHILE($2,$4); }

expr
        : T_CONST_INT
        | T_CONST_REAL
        | T_CONST_BOOL
        | T_LPAREN expr T_RPAREN
            { $$ = $2;}
        | T_IDENT
            { $$ = Absyn__VARIABLE($1);}
        | expr_bin
        | expr_un
        | expr_rel

expr_bin
        : expr T_PLUS expr
            { $$ = Absyn__BINARY($1, Absyn__ADD,$3);}
        | expr T_MINUS expr
            { $$ = Absyn__BINARY($1, Absyn__SUB,$3);}
        | expr T_TIMES expr
            { $$ = Absyn__BINARY($1, Absyn__MUL,$3);}
        | expr T_DIVIDE expr
            { $$ = Absyn__BINARY($1, Absyn__DIV,$3);}

expr_un
        : T_MINUS expr %prec T_UMINUS
            { $$ = Absyn__UNARY(Absyn__ADD,$2);}

expr_rel
        : expr T_LT expr
            { $$ = Absyn__RELATION($1,Absyn__LT,$3);}
        | expr T_LE expr
            { $$ = Absyn__RELATION($1,Absyn__LE,$3);}
        | expr T_GT expr
            { $$ = Absyn__RELATION($1,Absyn__GT,$3);}
        | expr T_GE expr

            { $$ = Absyn__RELATION($1,Absyn__GE,$3);}
        | expr T_NE expr
            { $$ = Absyn__RELATION($1,Absyn__NE,$3);}
        | expr T_EQ expr
            { $$ = Absyn__RELATION($1,Absyn__EQ,$3);}

%%
```

4.3.3.3 Main

```
package Main
  import PamDecl.ScanParse;
  import PamDecl.Eval;

type StringList = list<String>;

function mainprogram
  input StringList;
  output Boolean dummy;
algorithm
  ast := ScanParse.scanparse();
  ast := Eval.evalprog(ast);
  dummy := true; //?? should really call mainprogram recursively to have a loop
end mainprogram;

end Main;
```

4.3.3.4 ScanParse

```
package ScanParse
  import PamDecl.Absyn;

function scanparse
  output Absyn.Prog ast;
external "C";

end ScanParse;
```

4.3.3.5 scanparse.c

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

#include <stdio.h>
#include "rml.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 module */

void ScanParse_5finit(void) {}
```

```
/* The glue function */

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

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

  rmlA0=absyntree;
  RML_TAILCALLK(rmlSC);
}
RML_END_LABEL
```

4.3.3.6 makefile

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

# VARIABLES

SHELL = /bin/sh
LDLIBS = -lrml -ll # Order is essential; we want librml main, not libll!
LDFLAGS = -L$(RMLRUNTIME)/lib/plain/
CC = gcc
CFLAGS = -I$(RMLRUNTIME)/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.rml
        momc -c main.rml

# 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.rml
        momc -c absyn.rml

# ENVIRONMENTS

env.o:  env.c
env.c env.h:  env.rml
        momc -c env.rml

# EVALUATION

eval.o:  eval.c
eval.c eval.h:  eval.rml absyn.h env.h
        momc -c eval.rml

# 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 *~
```

4.3.4 Calling C from Meta-Modelica

The file `scanparse.rml` looks somewhat weird. It does not contain the usual module implementation section. In the makefile one also notices that it is not compiled using `momc`. Instead we supply the body for `scanparse.rml` through the file `scanparse.c`, which in turn is compiled in a regular way. This is the trick to use when wanting to call C from Meta-Modelica.

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 `scanparse.c` we need to implement the functions (functions) 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.rml`, which in this case does nothing.

If one want the function to fail call `RML_TAILCALLK(rmlFC)` or call `RML_TAILCALLK(rmlSC)` 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!");
```

4.4 Debugging Modelica Specifications

Even though Meta-Modelica is a specification language, it is common that specifications are erronous and therefore need to be debugged.

This section presents the interactive Meta-Modelica debugger functionality by showing a debugging session on a short Meta-Modelica example, together with a short overview of the debugger commands. The functionality of the debugger is illustrated using pictures from the Emacs debugging mode for Meta-Modelica (`Modelicadebug-mode`).

4.4.1 The Debugger Commands

The Emacs Modelica debug mode is implemented as a specialization of the Grand Unified Debugger (GUD) interface (`gud-mode`) from Emacs [??ref]. Because the Modelica debug mode is based on the GUD interface, some of the commands have the same familiar key bindings.

The actual commands sent to the debugger are also presented together with GUD commands preceded by the Modelica debugger prompt: `mdb@>`.

If the debugger commands have several alternatives these are presented using the notation: `alternative1|alternative2|....`

The optional command components are shown within square brackets: `[optional]`.

In the Emacs interface: `M-x` stands for holding down the `Meta` key (mapped to `Alt` in general) and pressing the key after the dash, here `x`, `C-x` stands for holding down the `Control (Ctrl)` key and pressing `x`, `<RET>` is equivalent with pressing the `Enter` key and `<SPC>` with pressing `Space` key.

4.4.1.1 Starting the Modelica Debugging Subprocess

The command for starting the Modelica debugger under Emacs is the following:

```
M-x Modelicadebug <RET> executable <RET>
```

4.4.1.2 Setting/Deleting Breakpoints

A part of a session using this type of commands is shown in Figure 4-1. The presentation of the commands follows later.

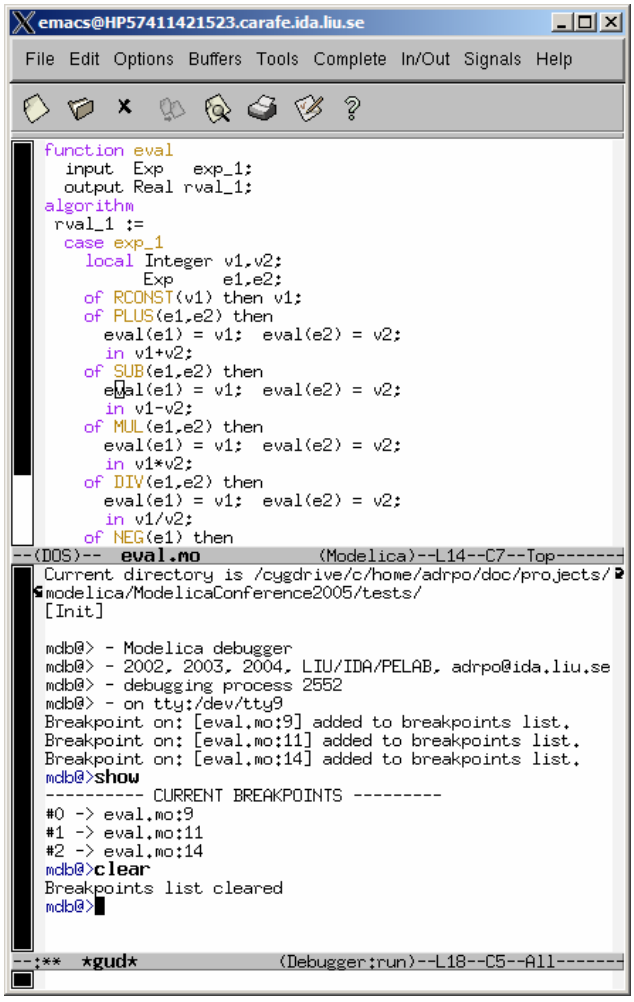


Figure 4-1. Using breakpoints.

To set a breakpoint on the line the cursor (point) is at:

```
C-x <SPC>
mdb@> break on file:lineno|string <RET>
```

To delete a breakpoint placed on the current source code line (`gud-remove`):

```
C-c C-d
```

```
C-x C-a C-d
mdb@> break off file:lineno|string <RET>
```

Instead of writing `break` one can use alternatives: `br|break|breakpoint`.

Alternatively one can delete all breakpoints using:

```
mdb@> cl|clear <RET>
```

Showing all breakpoints:

```
mdb@> sh|show <RET>
```

4.4.1.3 Stepping and Running

To perform one step (`gud-step`) in the Modelica code:

```
C-c C-s
C-x C-a C-s
mdb@> st|step <RET>
```

To continue after a step or a breakpoint (`gud-cont`) in the Modelica code:

```
C-c C-r
C-x C-a C-r
mdb@> ru|run <RET>
```

Examples of using these commands are shown in Figure 4-2. The example is the Exp1 calculator briefly described in Section 2.1.

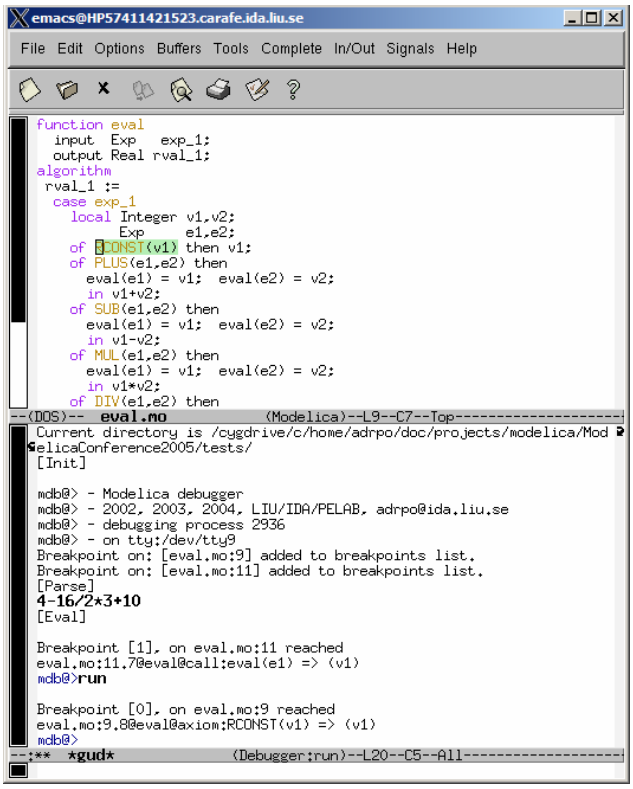


Figure 4-2. Stepping and running in the debugger.

4.4.1.4 Examining Data

There are no GUD keybindings for these commands but they are inspired from the GNU Project debugger (GDB) [ref??].

To print the contents/size of a variable one can write:

```
mdb@> pr|print variable_name <RET>
mdb@> sz|sizeof variable_name <RET>
```

at the debugger prompt. The size is displayed in bytes.

Variable values to be printed can be of a complex type and very large. One can restrict the depth of printing using:

```
mdb@> [set] de|depth integer <RET>
```

Moreover, we have implemented an external viewer written in Java called `DataViewer` to browse the contents of such a large variable. To send the contents of a variable to the external viewer for inspection one can use the command:

```
mdb@> bw|browse|gr|graph var_name <RET>
```

at the debugger prompt. The debugger will try to connect to the `DataViewer` and send the contents of the variable. The external data browser has to be started a priori. If the debugger cannot connect to the external viewer within a specified timeout a warning message will be displayed. A picture with the external `DataViewer` tool is presented in Figure 4-3:

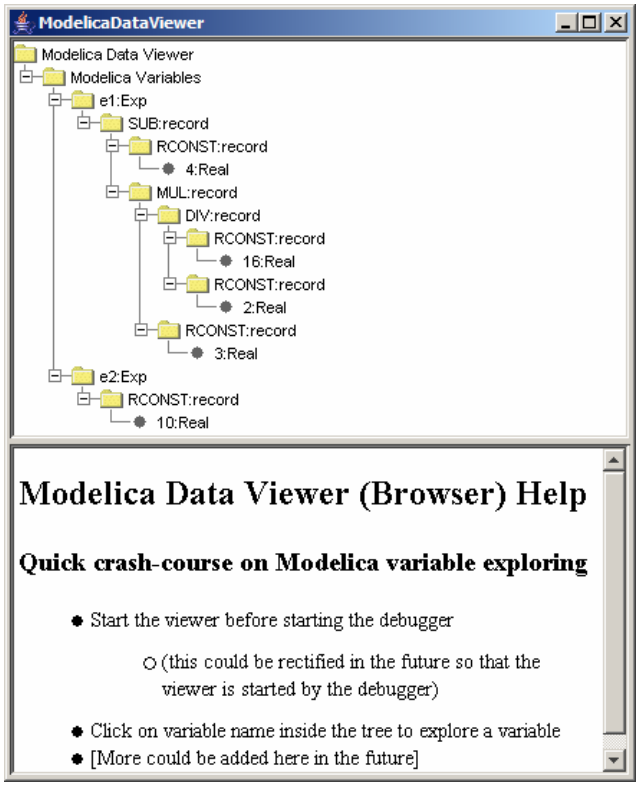


Figure 4-3. External browser/viewer for complicated data structures.

If the variable which one tries to print does not exist in the current scope (not a live variable) a notifying warning message will be displayed.

Automatic printing of variables at every step or breakpoint can be specified by adding a variable to a display list:

```
mdb@> di|display variable_name <RET>
```

To print the entire display list:

```
mdb@> di|display <RET>
```

Removing a display variable from the display list:

```
mdb@> un|undisplay variable_name <RET>
```

Removing all variables from the display list:

```
mdb@> undisplay <RET>
```

Printing the current live variables:

```
mdb@> li|live|livevars <RET>
```

Instructing the debugger to print or to disable the print of the live variable names at each step/breapoint:

```
mdb@> [set] li|live|livevars [on|off]<RET>
```

Figure 4-4 shows examples of some of these data examination commands within a debugging session:

Figure 4-4. Examining data in the debugger command window.

4.4.1.5 Additional commands

The stack contents (backtrace) can be displayed using:

```
mdb@> bt|backtrace <RET>
```

Because the contents of the stack can be quite large, one can print a filtered view of it:

```
mdb@> fbt|fbacktrace filter_string <RET>
```

Also, one can restrict the numbers of entries the debugger is storing using:

```
mdb@> maxbt|maxbacktrace integer <RET>
```

For displaying the status of the Modelica runtime:

```
mdb@> sts|stat|status <RET>
```

The status of the Meta-Modelica runtime comprises information regarding the garbage collector, allocated memory, stack usage, etc.

The current debugging settings can be displayed using:

```
mdb@> stg|settings <RET>
```

The settings printed are: the maximum remembered backtrace entries, the depth of variable printing, the current breakpoints, the live variables, the list of the display variables and the status of the runtime system.

One can invoke the debugging help by issuing:

```
mdb@> he|help <RET>
```

For leaving the debugger one can use the command:

```
mdb@> qu|quit|ex|exit|by|bye <RET>
```

A session using these commands is presented in Figure 4-5 below:

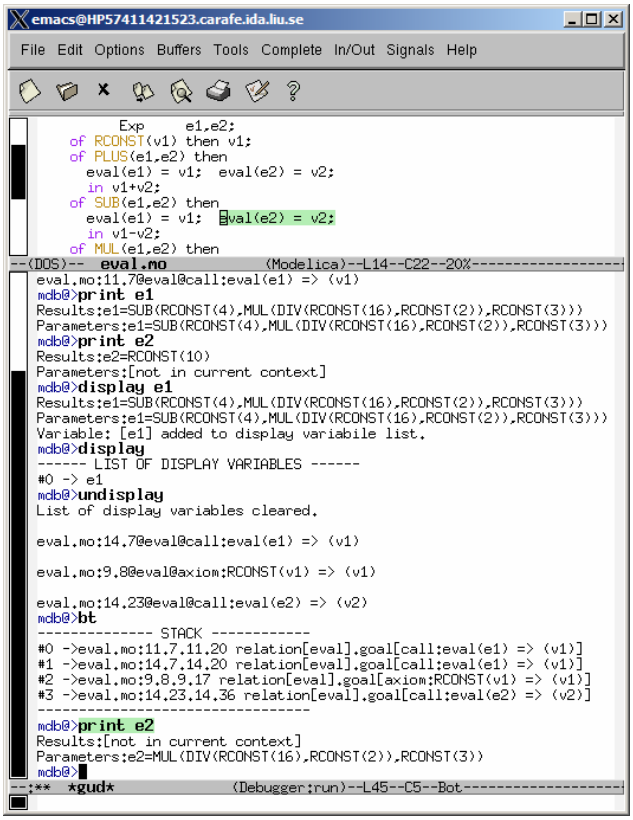


Figure 4-5. Additional debugger commands.

(BRK)

Chapter 5

Comprehensive Overview of the Current Meta-Modelica Subset

This chapter describes all the basic building blocks of Meta-Modelica such as characters and lexical units including identifiers, literals, and operators. Without question, the smallest building blocks in Meta-Modelica are single characters belonging to a character set. Characters are combined to form lexical units, also called tokens. These tokens are detected by the lexical analysis part of the Meta-Modelica translator. Examples of tokens are literal constants, identifiers, and operators. Comments are not really lexical units since they are eventually discarded. On the other hand, comments are detected by the lexical analyzer before being thrown away.

The lexical units are combined to form even larger building blocks such as expressions according to the rules given by the expression part of the Meta-Modelica grammar.

5.1 Meta-Modelica Constructs to be Depreciated

The current Meta-Modelica subset contains several constructs which will eventually be depreciated, i.e. removed, from the Meta-Modelica language. They are needed right now, before compiler support for better alternatives has been implemented. The constructs to be depreciated are the following:

- matchcontinue-expressions will be replaced by match-expressions with guards.
- Real number arithmetic operators containing a dot (+., -., *., /., etc.) will be replaced by ordinary overloaded arithmetic operators (+, - ,* , /, etc.)
- The equality(...) operator will be removed.
- etc...

5.2 Meta-Modelica Constructs not yet Fully Supported

The following constructs are not yet fully implemented:

- match-expressions currently work the same way as matchcontinue-expressions.
- Guards with the guard keyword are not yet supported in match/matchcontinue-expresions.
- Named argument to functions and constructors are not yet supported.
- Named arguments in constructor-calls in patterns are not yet supported.

- etc...

5.3 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, which corresponds to the first 256 characters of the 16-bit Unicode character set. Most of the first 128 characters of Latin-1 are equivalent to the 7-bit ASCII character set.

5.4 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:

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

5.5 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.

5.5.1 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 Meta-Modelica 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"
```

5.5.2 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. Package names are described in more detail in ??.

5.5.3 Meta-Modelica Keywords

The following Meta-Modelica *keywords* are reserved words and may not be used as identifiers:

```
_                           and             annotation      block
                            case            constant
else                                                        end
equality        equation                    external        false
failure                                     function        if
input           list        local           match           matchcontinue
not             or                          output          package
                            protected       public          record
                                            then            true
tuple           type        uniontype
```

5.6 Predefined Types

The predefined built-in based types of Meta-Modelica 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 here 31 bit integer)
`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 Meta-Modelica, `Real` corresponds to `double` and `Integer` corresponds to `int`.

5.6.1 Literal Constants

Literal constants are unnamed constants that have different forms depending on their type. Each of the predefined types in Meta-Modelica 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.

5.6.2 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
```

5.6.3 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 Integer type in Meta-Modelica. However, the `Long` builtin type supports 64-bit 2-complement integers.

5.6.4 Booleans

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

5.6.5 Strings

String literals appear between double quotes as in `"between"`. Any character in the Meta-Modelica 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.

5.6.6 Array Literals

Array literals can be expressed using the array constructor { } or `array(...)`. For example, the following are one-dimensional array constants, i.e., vector literals:

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

Two-dimensional array constants, i.e., matrix literals, may occur as arrays of arrays:

```
{{1,2},{3,4}}
```

5.6.7 List Literals

List literals can be expressed using the list or array constructor { }, or by `list(...)`. For example, the following are one-dimensional list constants:

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

The { } constructor can be used construct either arrays or lists. The type context determines which interpretation is chosen. It is possible to unambigously specifiy the creating of a list value by using the `list(...)` builtin function:

```
list(1,2,3),     list(3.14, 58E-6)
```

5.6.8 Record Literals

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

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

5.7 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	`.`	`a.b`
postfix function call	(*function-arguments*)	`sin(4.36)`
array or list construction	{*expressions*} array(expressions) list(expressions)	`{2,3}`
integer or real multiplicative	`*  /  *.  /.`	`2*3  2/3  2.1 *. 3.2`
integer or real additive	`+  -  +`*expr* `-`*expr* `+. -.  +.` *expr* `-.` *expr*	`a+b, a-b, +a, -a` `a+.b, a-.b, +.a, -.a`

integer or real relational
...
unary negation
logical and
logical or
conditional expression
list element concatenation
	`<  <=  >  >=  ==  <>`
	`<.  <=.  >. >=.  ==.  <>.`
	`not` *expr*
	`and`
	`or`
	`if` *expr* `then` *expr* `else` *expr*
	`"a"::{"b","c"}` => `{"a","b","c"}`
		`a<b, a<=b, a>b, ...`
		`a<.b, a<=.b, a>.b, ...`
		
		`not b1`
		`b1 and b2`
		`b1 or  b2`
		`if b then 3 else x`
		`"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 equality operator, equality(expr1 = expr2), 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:

`sin`(*u*)	sine
`cos`(*u*)	cosine
`tan`(*u*)	tangent (*u* shall not be: ..., $-\pi/2,\ \pi/2,\ 3\pi/2, \ldots$)
`asin`(*u*)	inverse sine $(-1 \le u \le 1)$

`acos`(*u*)	inverse cosine $(-1 \le u \le 1)$
`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:

```
bool_and, bool_or, bool_not
```

Integer operations:

```
int_add, int_sub, int_mul, int_div
int_mod, int_abs, int_neg, int_max, int_min
int_lt, int_le, int_eq, int_ne, int_ge, int_gt, int_real, int_string
```

Real number operations:

```
real_add, real_sub, real_mul, real_div
real_mod, real_abs, real_neg, real_max, real_min
real_lt, real_le, real_eq, real_ne, real_ge, real_gt, real_int, real_string
real_cos, real_sin, real_atan, real_exp, real_ln, real_floor, real_int, real_pow
```

String operations:

```
string_length, string_nth, string_append
string_int, string_list, list_string
```

5.8 Arithmetic Operators

Meta-Modelica supports five binary arithmetic operators in both integer and real variants. The real number operators currently contain a dot.

```
^ .          Exponentiation
*   *.       Multiplication
/   /.       Division
+   +.       Addition
-   -.       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.

5.8.1 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 Meta-Modelica compilers.

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 Meta-Modelica 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.

5.8.1.1 Long Integers

?? fill in

5.8.2 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 that 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.

5.9 Equality, Relational, and Logical Operators

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

>	greater than
>=	greater than or equal
<	less than
<=	less than or equal to
==	equality within expressions
<>	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.

=	equality within equations
=	assignment of named arguments at function call

The following logical operators are defined:

`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. See also Section **Error! Reference source not found.**, page **Error! Bookmark not defined.**, for guarding evaluation.

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

5.9.1 String Concatenation

The + operator is also a built-in string concatenation operator in Standard Modelica, both for string variables and literal string constants. For example, long comment strings can be constructed using the + operator for concatenation of string constants, e.g.:

```
Real longval = 1.35E+300  "This is" + " a " + "rather " + " long comment";
```

Another example using string variables and string literals in expressions returning string values:

```
String val1 = "This is";
String val2 = " a ";
String concatvalue = val1 + val2 +"rather " + " long string";
// The value becomes: "This is a rather long string"
```

5.9.2 The Conditional Operator—if-expressions

The conditional operator in Meta-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);
```

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

5.10 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:

`failure(...)`	Fill in
`equality(...)`	Fill in??
`bool_success(...)`	Fill in
`list()`	Fill in
`array(...)`	??Fill in

5.11 Order of Evaluation

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

5.12 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.

5.12.1 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. Meta-Modelica 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.

5.12.1.1 Implicit Type Conversions

Sometimes a type can be converted without any explicit action from the Modelica programmer. The only case in full Modelica when this happens is *implicit conversion* of integer operands when used together with floating point operands in an expression. However, in the current Meta-Modelica, all type conversions must by explicit..

5.12.1.2 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`. (?? add stuff)

5.13 Global Constant Variables

Global constants can be declared in Meta-Modelica through the **`constant`** keyword, e.g. as below where the `init_env` variable is set to the empty list:

```
constant init_env = {}
```

5.14 Types

The Meta-Modelica 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.

5.14.1 Primitive Data Types

The Meta-Modelica language provides a basic set of primitive types found in most programming languages:

- `Boolean`—booleans, e.g. `true/false`.
- `Integer`—integers, e.g. `-123`. (?? 31-bit integers in the current Meta-Modelica version; Long integers are also available ?)
- `Real`—double-precision IEEE floating point numbers, e.g. `3.2E5`.
- `String`—strings of characters, e.g. `"Linköping"`.

5.14.2 Type Name Declarations

Alternate names for types in Meta-Modelica can be introduced through the **`type`** declaration, e.g.:

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

5.14.3 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>;
```

5.14.4 Tagged Union Types for Records, Trees, and Graphs

The **`uniontype`** declaration in Meta-Modelica 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 x1;  end INT;
  record RATIONAL  Integer x1;  Integer x2;  end RATIONAL;
  record REAL      Real x1;  end REAL;
  record COMPLEX   Real x1;  Real x2;  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 Meta-Modelica 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 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;
```

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)
```

However, building circular structures is not possible because of the declarative side-effect free nature of Meta-Modelica. 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.

5.14.5 Parameterized Data Types

A *parameterized data type* in Meta-Modelica 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 Meta-Modelica language provides three kinds of parameterized types:

- Lists – the **`list`** identifier, parameterized in terms of the list element type.
- Vectors – the **`array`** identifier, parameterized in terms of the vector 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 Meta-Modelica 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 Meta-Modelica can be represented by arrays of elements of tagged union types, where each "type" in the union type is denoted by a different tag.

5.14.5.1 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 Meta-Modelica list operations are briefly described by the following examples; see Appendix ??B for type signatures of these functions:

```
list_append({2,3},{4,5})  => {2,3,4,5}
list_reverse({2,3,4,5})   => {5,4,3,2}
list_length({2,3,4,5})    => 4
list_member(3, {2,3,4,5}) => true
list_get({2,3,4,5}, 4)    => 5     // First list element is numbered 1
list_delete({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 Meta-Modelica **`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 Meta-Modelica list type with an unspecified element type `TYpe_elemtype` which is a type parameter (type variable) of the list. Type variable names in Meta-Modelica are declared as replaceable types.

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

Lists in the Meta-Modelica 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.

5.14.5.2 Arrays and Vectors

An Meta-Modelica vector is a sequence of elements, all of the same type. The main advantage of a vector compared to a list is that an arbitrary element of a vector can be accessed in constant time by a vector indexing operation on a vector and an integer denoting the ordinal position of the element.

Constructing vectors is rather clumsy in Meta-Modelica. First a list has to be constructed which then is converted to a vector, e.g.: (?? update)

```
list_vector({2,4,6,8}) => vec
```

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

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

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

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

Getting the length of vector `vec`:

```
vector_length(vec) => 4
```

Named array types can of course be declared using the **`type`** construct, e.g. as in the declaration of a one-dimensional vector 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 vector 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;
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 vector operations, also presented in Appendix B??. The following are the length and indexing signatures:

```
function vector_length  "Compute the length of a vector"
  input Type_a[:]  in_vec;
  output Integer   out_length;
protected
  replaceable type Type_a;
end vector_length;

function vector_get  "Extract (indexed access) a vector element from the vector"
  input Type_a[:]  in_vec;
  output Type_a    out_element;
protected
  replaceable type Type_a;
end vector_get;
```

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

```
function vector_list  "convert from vector to list"
  input Type_a[:]      in_vec;
  output list<Type_a>  out_lst;
protected
  replaceable type Type_a;
end vector_list;

function list_vector  "Convert from list to vector"
  input list<Type_a>  in_lst;
  output Type_a[:]    out_vec;
protected
  replaceable type Type_a;
end list_vector;
```

5.14.5.3 Option Types

Option types have been introduced in Meta-Modelica 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 Meta-Modelica union type with the two constructors `NONE()` and `SOME()`:

```
uniontype Option
  replaceable type Type_a;
  record NONE  end NONE;
  record SOME  Type_a x1;  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.

5.15 Meta-Modelica Functions

We have already used Meta-Modelica 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 Meta-Modelica 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 ??? for a discussion of these functions.

5.15.1 Function Declaration

The body of a Meta-Modelica function is a kind of algorithm section that contains procedural algorithmic code to be executed when the function is called. Formal parameters are specified using the `input` keyword, whereas results are denoted using the `output` keyword. This makes the syntax of function definitions quite close to Modelica class definitions.

The structure of a typical function declaration is sketched by the following schematic function example:

```
function <functionname>
  input  TypeI1 in1;
  input  TypeI2 in2;
  input  TypeI3 in3 := <default expr> "Comment" annotation(...);
  ...
  output TypeO1 out1;
  output TypeO2 out2 := <default expr>;
  ...
protected
  <local variables>
  ...
algorithm
  ...
   <statements>
  ...
end <functionname>;
```

Optional explicit default values can be associated with any input or output formal parameter through declaration assignments. Such defaults are shown for the third input parameter and the second output parameter in our example. Comment strings and annotations can be given for any formal parameter declaration, as usual in Meta-Modelica declarations.

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

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

5.15.2 Current Restrictions of Meta-Modelica Functions

Only two supported forms of functions are supported by the current version of the Meta-Modelica compiler:

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

An example of the first kind:

```
function eval_stmt_list "Evaluate a list of statements in an environment.
                         Pass environment forward"
  input Env.Env in_env;
  input Absyn.StmtList in_stmtlist;
  output Env.Env out_env;
algorithm
  out_env :=
  match (in_env,in_stmtlist)
    local
      type Env_BindList = list<Env.Bind>;
      Env_BindList env;
    case (env,{}) then env;
    case (env, s :: ss)
      equation
        env1 = eval_stmt(env, s);
        env2 = eval_stmt_list(env1, ss); then env2;
  end match;
end eval_stmt_list;
```

An example of the second kind:

```
function input_item  "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 input_item;
```

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- or match-continue expression, formal input parameters may only be referenced directly after the `match`/`matchcontinue` keyword, e.g. `match (in_x, in_y)...` or `match in_z ...`, 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.

5.15.3 Returning Single or Multiple Function Results

A function with one output formal parameter always returns a *single result*. Our previously presented example functions `polynomialEvaluator` and `realToString` are single result functions.

However, a function with *more* than one output formal parameter has *multiple results*. An example is the function `pointOnCircle` below, which computes the cartesian coordinates of a point located at a certain angle on a circle with a specific radius. The Cartesian coordinates are returned via the two result variables `x` and `y`.

```
function pointOnCircle  "Computes cartesian coordinates of a point"
  input  Real angle  "angle in radians";
  input  Real radius;
  output Real x;    // 1:st result formal parameter
  output Real y;    // 2:nd result formal parameter
algorithm
  x := radius*cos(phi);
  y := radius*sin(phi);
end pointOnCircle;
```

If we call a function with just one result we can put the call anywhere within an expression. This is also the case when calling a function with multiple results if we only want to access its first result.

On the other hand, if we wish to call a Modelica function with multiple results and want to obtain more than the first result, there are just two syntactic forms to choose from depending on whether the function call should occur in an equation section or in an algorithm section: one equation form and one statement form, as specified below:

```
(out1,out2,out3,...) = function_name(in1, in2, in3, in4, ...);  // Equation
(out1,out2,out3,...) := function_name(in1, in2, in3, in4, ...); // Statement
```

The left-hand side of both the equation and the assignment statement contains a parenthesized, comma-separated list of variables receiving the results from the function call. A called function with *n* results can have at most *n* receiving variables on the left-hand side. Fewer than *n* receiving variables means that some function results are discarded.

For example, when calling our example function `pointOnCircle` with two receiving variables `px` and `py` on the left-hand side, such calls can appear as follows:

```
(px,py)  = pointOnCircle(1.2, 2);    // Equation form
(px,py) := pointOnCircle(1.2, 2);    // Statement form
```

Any kind of variable of compatible type is allowed in the list on the left-hand side, e.g. array elements:

```
(arr[1],arr[2]) := pointOnCircle(1.2, 2);
```

To summarize, the following rules apply when returning results from a multiple-result function:

- The variables in the list on the left-hand side of the equation or assignment containing the call are associated with the returned function results according to the order of the variables in the list and the corresponding declaration order of the output result variables in the function.
- As in any standard equation or assignment, the type of each variable on the left-hand side must be compatible with the type of the corresponding function result on the right-hand side, with or without type coercion.

5.15.4 Builtin Functions

A number of "standard" builtin primitives are provided by the Modelica standard library—in a module called `???`. Examples are `int_add`, `int_sub`, `string_append`, `list_append`, etc. A complete list of these primitives can be found in ??Appendix B.

5.15.5 Special Properties of Modelica Match Expressions

Two important properties of Meta-Modelica functions are however absent for ordinary functions:

- Functions in Meta-Modelica can fail or succeed.
- Retry is supported between rules in a matchcontinue 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 rule, or in one of the functions directly or indirectly called via the local equations of the rule, and a matchcontinue-expression is used, Meta-Modelica will backtrack (i.e., undo) the part of the "execution" which started from this rule, and automatically *continue* with the next rule (if there is one) in top-down, left-to-right order. If no rule 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 rules of the specification.

5.15.6 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 Meta-Modelica function. One or more results can be returned from such a call. The issues are discussed in some detail in the following sections.

5.15.6.1 Multiple Arguments and Results

A Meta-Modelica 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.

5.15.6.2 Tuple Arguments and Results from Relations

We just noted that a Meta-Modelica function can have multiple arguments and results. This should not be confused with the case where a Modelica tuple type (see Section 5.14.3) 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>  in_val;
  output tuple<Integer,Integer> out_val;
algorithm
  out_val :=
  match in_val
    local Integer x1,x2;
    case (x1,x2) then (x1+1,x2+1);
  end match;
end incrementpair;
```

For example, the call:

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

gives the result:

```
(3,4)
```

5.15.6.3 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.

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 list_map /*
 ** Takes a list and a function over the elements of the lists, which is applied
 ** for each element, producing a new list.
 ** For example  list_map({1,2,3}, int_string) => { "1", "2", "3"}
```

```
  */
  input list<Type_a> in_aList;
  input FuncType     in_func;
  output list<Type_b> out_bList;
protected
  replaceable type Type_a;
  replaceable type Type_b;
  function FuncType
    replaceable type Type_b;
    input Type_a in_a;
    output Type_b out_b;
  end FuncType;
algorithm
  out_bList:=
  match (in_aList,in_func)
    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 = list_map(rest, fn); then first_1 :: rest_1;
  end match;
end list_map;


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

5.16 Variables and Types in Functions

Except for global constants, Meta-Modelica variables only occur in functions. Types, including parameterized types, can be explicitly declared in Meta-Modelica function type signatures.

5.16.1.1 Type Variables and Parameterized Types in Relations

We have already presented the notion of parameterized list, vector, and option types in Section 5.14.5. Type variables in Meta-Modelica can only appear in function signatures.

For example, the `tuple2_get_field1` 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.:

```
tuple2_get_field1(("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 tuple2_get_field1 "
 ** Takes a tuple of two values and returns the first value.
 ** For example,
 ** tuple get_field1((true,1)) => true
 *"
  input tuple<Type_a,Type_b> in_tuple;
  output Type_a out_Type_a;
protected
  replaceable type Type_a;
  replaceable type Type_b;
algorithm
  out_Type_a :=
  match (in_tuple)
    local Type_a a;
    case (a,_) then a;
  end match;
end tuple2_get_field1;
```

5.16.1.2 Local Variables in Match-Expressions in Functions

Variables in Meta-Modelica functions consisting of match-expressions are normally introduced at the beginning of a match-expression or in math-expression rules and have a scope throughout the rule. The only exception are global constants. There are three kinds of local variables for values, as well as type variables which are introduced through replaceable type declarations:

- *Pattern local variables*, which are given values in patterns to be matched.
- *Ordnary local variables*, which occur on the left hand side of equality signs, e.g.: variable = *expression*. Result variables can be regarded as a special case of pattern variables, for the trivial pattern consisting of the variable itself.
- *Type variables*, which are declared using replaceable type and introduced in the function protected section.

For example, in the function `list_thread` below, `Type_a` is a type variable for the type of elements in the list, `fa`, `rest_a`, `fb`, `rest_b` are pattern variables in the pattern `list_thread(fa::rest_a, fb::rest_b)`:

```
function list_thread
 "Takes two lists of the same type and threads them together.
  For example,  list_thread({1,2,3},{4,5,6}) => {4,1,5,2,6,3}
 "
  input list<Type_a> in_List1;
  input list<Type_a> in_List2;
  output list<Type_a> out_List;
protected
  replaceable type Type_a;
algorithm
  out_List:=
```

```
  match (in_List1,in_List2)
    local
      list<Type_a> rest_a,rest_b;  Type_a fa,fb;
    case ({},{}) then {};
    case (fa :: rest_a, fb :: rest_b)
      then fa :: fb :: list_thread(rest_a, rest_b);
  end match;
end list_thread;
```

5.16.2 Function Failure Versus Boolean Negation

We have previously mentioned that Meta-Modelica 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 rule 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 rule in a match-expression to fail immediately. (?? A better semantics would be to 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 bool_not
  input  Boolean in_bool;
  output Boolean out_bool;
algorithm
  out_bool := if in_bool == true then false else true;
end bool_not;
```

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 rule to fail. The Modelica `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.

5.16.3 Forms of Equations in Rules

The local equations in a Meta-Modelica rule are currently restricted to having the following forms, where `func_name` is the name of a function; see also the Meta-Modelica grammar in ??Appendix ??, and `expr` may contain constants, variables, constructor calls, and operators, but currently not functions:

- `expr = func_name(...)`
- `func_name(...)`

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

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

5.17 Pattern-Matching

Pattern-matching on instances of structured data types is one of the central facilities provided by Meta-Modelica, which significantly contributes to the elegance and ease with which many language aspects may be specified. The pattern matching in Meta-Modelica is very close to similar facilities in many functional languages.

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.

5.17.1 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 of a conclusion in the rule below:

```
match argument
  local Integer x;
case  INT(x) ...
```

This means that `argument` is matched using the pattern `INT(x)`. If there is a match, the rule 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 rule, 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)`.

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

```
match ...
    local  Integer u;  String w;
  case ...
    equation
      (u,w) = ...;
```

If the right-hand side of the local equation produces the tuple `(55,"Test)`, and `u` and `w` are unbound, then the match to the pattern `(u,w)` will succeed by binding `u` to 55 and `w` to `"Test"`.

5.17.2 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 rule below after the **`then`** keyword:

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

If the rule 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 rule belongs:

```
(44, {5,"Hello"}, INT(77))
```

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 Meta-Modelica function for computing the sum of integers can be expressed as follows:

```
function sum
  input Integer in_i;
  input Integer in_n;
```

```
  output Integer out_res;
algorithm
  out_res :=
  matchcontinue (in_i,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 Meta-Modelica language is available, the above `sum` function can be expressed more concisely:

```
function sum
  input Integer i;
  input Integer n;
  output Integer out_res;
algorithm
  out_res := 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 Meta-Modelica functions:

```
function sum
  input Integer i;
  input Integer n;
```

```
  output Integer out_res;
algorithm
  out_res := sumTail(i,n,0);
end sum;

function sumTail
  input Integer in_i;
  input Integer in_n;
  input Integer in_sumSoFar;
  output Integer out_res;
algorithm
  out_res :=
  matchcontinue (in_i, in_n, in_sumSoFar)
    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 rule.

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 out_res;
algorithm
  out_res := 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 `list_thread` function from Section 5.16.1.2, called `list_thread_tail`:

```
list_thread(a,b) = list_thread_tail(a,b,{})
```

We have introduced an accumulating parameter as the third argument of `list_thread_tail`, e.g.:

```
list_thread_tail({1,2,3},{4,5,6},{}) => {4,1,5,2,6,3}
```

Its definition follows below:

```
function list_thread_tail
 "Takes two lists of the same type and threads them togheter.
  For example,  list_thread({1,2,3},{4,5,6}) => {4,1,5,2,6,3}
 "
  input list<Type_a> in_List1;
  input list<Type_a> in_List2;
  input list<Type_a> in_accumlst;
```

```
  output list<Type_a> out_List;
protected
  replaceable type Type_a;
algorithm
  out_List:=
  match (in_List1,in_List2,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 list_thread_tail(rest_a, rest_b, fa :: fb :: accumlst);
  end match;
end list_thread_tail;
```

6.1.2 Using Side Effects

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 rules, 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 rule (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 rule 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 rule 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 `choose_foo`.

```
function choose_foo
  input  Real in_x;
  output Real out_y;
```

```
algorithm
  out_y :=
  matchcontinue 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 choose_foo;

function foo
  input  Real in_x;
  output Real out_y;
protected
  Real x,y,z;
algorithm
  (z,y) := choose_foo(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 Meta-Modelica 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 Meta-Modelica.

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 Meta-Modelica Rules

Below we present a number of issues regarding the semantics and usage of Meta-Modelica rules.

6.2.1 Logically Overlapping Rules

A programming language specification in Meta-Modelica are often written in such a way that the local equations of different rules 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 Meta-Modelica will resolve such ambiguities since the first matching rule will always return in the interval $3 \leq x < 5$. Thus, the first rule giving the value x+10 will be selected.

There is one rather common case where logically overlapping rules together with Meta-Modelica's search rule of rule matching top-down, left-to-right, can be used to advantage, to allow more *concise* and easily readable specifications. The rules can be ordered such that rules with more *specific* conditions appear first, and more *general* rules which may logically overlap some previous rules appear later.

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

Anyway, the style of specification with more specific conditions first and more general rules later makes sense from a logical point of view when interpreted together with Meta-Modelica'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 rules is changed. Such a set of rules can be converted to a semantically equivalent set of clumsier non-overlapping rules. Negated conjunctions must then be added to overlapping rules.

6.2.2 Using else Default Rule in Match-Expressions

There is a common situation 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 rules—return `false`.

```
function isunfold
  input  Ty      in_node;
  output Boolean out_res;
algorithm
  out_res :=
    match in_node
      case UNFOLD(_) then true;
      case ARITH(_) then false;
      case PTR(_) then false;
      case ARR(_,_) then false;
      case REC(_) then false;
    end match;
end function;
```

A more concise specification of this function can be obtained by adding a default rule at the end of the match-expression with a general pattern that matches all cases returning the same default result. The top-down, left-to-right search rule in match-expressions ensures that the special cases will match if they occur—before the default case which always matches. The logical specification purist will unfortunately regard such a specification as logically incorrect because of the overlap. Meta-Modelica solves this problem by providing an explicit default else-rule in match-expressions, as in the example below:

```
function isunfold
  input  Ty      in_node;
  output Boolean out_res;
algorithm
  out_res :=
    match in_node
      case UNFOLD(_) then true;
      else then false;
    end match;
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.

Meta-Modelica supports a limited form of higher-order programming: functions can be passed as arguments to other functions, but cannot be returned as values or directly assigned as values.

We give three examples of higher-order Meta-Modelica functions that take another function as a parameter, and a function that can be used as a conditional expression (`if`) construct within a single Meta-Modelica rule. The functions are the following:

- `if_`
- `list_reduce`
- `list_map`
- `list_fold`

The `if_` function makes it possible in many cases to avoid having the then-part and the else-part as separate rules.

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 in_boolean1;
  input Type_a in_type_a2;
  input Type_a in_type_a3;
  output Type_a out_type_a;
protected
  replaceable type Type_a;
algorithm
  out_type_a:=
  match (in_boolean1,in_type_a2,in_type_a3)
    local Type_a r;
    case (true,r,_) then r;
    case (false,_,r) then r;
  end match;
end if_;
```

The `list_reduce` 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.

```
list_reduce({1,2,3},int_add) => 6

function list_reduce
  input VType_aList in_vtype_alist;
  input FuncType in_func;
  output Type_a out_type_a;
protected
  replaceable type Type_a;
  type VType_aList = list<Type_a>;
```

```
  function FuncTyp
    input Type_a in_type_a1;
    input Type_a in_type_a2;
    output Type_a out_type_a;
  end FuncType;
algorithm
  out_type_a:=
  match (in_vtype_alist,in_func)
    local
      Type_a e,res,a,b,res1,res2;
      FuncType r;
      VType_aList 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 = list_reduce(xs, r);
        res = r(res1, res2); then res;
  end match;
end list_reduce;
```

The `list_map` 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, `int_string` has the signature: (`int => string`)

```
list_map({1,2,3}, int_string) => { "1", "2", "3"}
```

```
function list_map
  input VType_aList in_vtype_alist;
  input FuncType in_func;
  output VType_bList out_vtype_blist;
protected
  replaceable type Type_a;
  type VType_aList = list<Type_a>;
  function FuncType
    input Type_a in_type_a;
    output Type_b out_type_b;
   protected
    replaceable type Type_b;
  end FuncType;
  replaceable type Type_b;
  type VType_bList = list<Type_b>;
algorithm
  out_vtype_blist:=
  match (in_vtype_alist,in_func)
    local
      Type_b f_1;
      VType_bList r_1;
      Type_a f;
      VType_aList r;
      FuncType fn;
    case ({},_) then {};
```

```
    case (f :: r,fn)
      equation
        f_1 = fn(f);
        r_1 = list_map(r, fn); then f_1 :: r_1;
  end match;
end list_map;
```

The `list_fold` 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. `list_fold` will call the passed function for each element in a sequence, adding to the accumulating parameter value.

```
list_fold({1,2,3},int_add,2) =>  8

int_add(1,2) => 3, int_add(2,3) => 5, int_add(3,5) => 8

function list_fold
  input VType_aList in_vtype_alist;
  input FuncType in_func;
  input Type_b in_type_b;
  output Type_b out_type_b;
protected
  replaceable type Type_a;
  type VType_aList = list<Type_a>;
  function FuncType
    input Type_a in_type_a;
    input Type_b in_type_b;
    output Type_b out_type_b;
   protected
    replaceable type Type_b;
  end FuncType;
  replaceable type Type_b;
algorithm
  out_type_b:=
  match (in_vtype_alist,in_func,in_type_b)
    local
      FuncType r;
      Type_b b,b_1,b_2;
      Type_a l;
      VType_aList lst;
    case ({},r,b) then b;
    case (l :: lst,r,b)
      equation
        b_1 = r(l, b);
        b_2 = list_fold(lst, r, b_1); then b_2;
  end match;
end list_fold;
```

(BRK)

Index