Modelica Text Template Language Susan Users Guide

2011-04-07 version 0.3

Summary

This is the Modelica text template language Susan Users guide.

Revisions

v0.3 2011-04-07 v6	Pavol Privitzer. Introduced new Section 2.9.1 on interface package import, Section 2.10 on template file import into another template package, and Section 2.11 on template error handling. Also added description of countEmpty and separateEmpty options in Section 2.8.2. Also updated Section 2.8.3 about empty and countEmpty options.
v0.3 2011-03-31 v6	Peter Fritzson. Some corrections after Stavåkers comments. Also update of indexby to hasindex changes and from to fromindex after info from Per and Pavol. Included unfinished template import Section 2.9.1 and error handling Section 2.11 after information from Martin.
v0.2 2010-04-21 v5	Peter Fritzson. Fixed some errors found by Martin and Anton
v0.2 2010-04-19 v4	Peter Fritzson. Restructured the document and made the first full draft.
v0.1 2010-04-01 v1	Peter Fritzson. Started the work.

Table of Contents

Chapter 1

Background

1.1 Intended Use

The Modelica template language Susan is intended to simplify and decrease costs of implementation and maintenance of code generators, unparsers, XML emitters, etc. from OpenModelica intermediate code (AST, lower level tree IR, all in MetaModelica) to text. The generated code can for example be in C, C#, Java, XML, or some other language.

1.2 User Profile

The intended user knows Modelica and MetaModelica, C/Java (as most people), and usually is contributing to the OpenModelica compiler. The user will typically not have any knowledge of other text template languages.

1.3 Background and Motivation

Traditionally, models in a modeling language such as Modelica are primarily used for simulation. However, the modeling community needs not only tools for simulation but also languages and tools to create, query, manipulate, and compose equation-based models. Examples are parallelization of models, optimization of models, checking and configuration of models, generation of program code, documentation and web pages from models.

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

An alternative idea that already to some extent has been explored in MetaModelica [9][22] is to add extensibility features to the modeling language. For example, a model package could contain model analysis and translation features that therefore are not needed in the model compiler. An example is a PDEs discretization scheme that could be expressed in the modeling language itself as part of a PDE package instead of being added internally to the model compiler.

Such transformation and analysis operations typically operate on abstract syntax tree (AST) representations of the model. Therefore the model needs to be converted to tree form by *parsing* before transformation, and later be converted back into text by the process of *unparsing*, also called *pretty printing*.

The MetaModelica work is primarily focused on mechanisms for mapping/transforming models as structured data (AST) into structured data (AST), which is needed in advanced symbolic transformations and compilers.

However, there is an important *subclass* of problems mapping structured data (AST) representations of models into text. Unparsing is one example. Generation of simulation code in C or some other language from a flattened model representation is another example. Yet another use case is model or document generation based on text templates where only (small) parts of the target text needs to be replaced.

We believe that providing a template language for Modelica may fulfill a need for an easier-to-use approach to a class of applications in model transformation based on conversion of structure into text. Particularly, we want an operational template language that enables to retarge thet OpenModelica compiler simply by specifying a package of templates for the new target language.

1.4 Definition of Text Template Language

In this section we try to be more precise regarding what is meant by the notion of text template language, template and template function.

Definition 1. Template Language. A template language is a language for specifying the transformation of structured data into a textual target data representation, by the use of a parameterized object "the template" and constructs for specifying the template and the passing of actual parameters into the template.

One could generalize the notion of template language to cover target language representations that are not textual. However, in the following we only concern ourselves with textual template languages.

Definition 2. Template and Template Function. A *template function* is a function from a set of attributes/parameters to a textual data structure.

A *template* is a text string with holes in it. The holes are filled by evaluating expressions that are converted to text when evaluating the template body. More formally, we can use the definition from [18] (slightly adapted):

A template function is a function that maps a set of attributes to a textual data structure. It can be specified via an alternating list of text strings, t_i, and expressions, e_i, that are functions of attributes a_i:

$$F(a_1, a_2, ..., a_m) ::= t_0\, e_0 ... t_i\, e_i\, t_{i+1} ... t_n\, e_n\, t_{n+1}$$

where t_i may be the empty string and e_i is restricted computationally and syntactically to enforce strict model-view separation, see Section 1.5 and [19]. The e_i are distinguished from the surrounding text strings by bracket symbols. Some design alternatives are angle brackets `<...>`, dollar sign `$...$`, or combined `<%...%>` as in Susan. Evaluating a template involves traversing and concatenating all t_i and e_i expression results.

Definition 3. Textual Data Structure. A textual data structure has text data such as strings of characters as leaf elements. Examples of textual data are: a string, a list (or nested list structure) of strings, an array of strings, or a text file containing a single (large) string. A textual data structure should efficiently be able to convert (flattened) into a string or text file. In the Modelica template language Susan, the textual data structure is the predefined `Text` type.

1.5 Design Principles for the Modelica Template Language Susan

As mentioned, a text template is essentially a text with holes. The holes are filled by implicit conversion of an data structure to text. In the template language, a template is a function from arguments to a textual data structure.

The template language design could either be a domain specific extension of an existing language (MetaModelica), an extended subset of an existing language, or a new domain specific language, DSL. The current design is the latter.

The current design and syntax is influenced by other template languages, especially the Stringtemplate language, as well as by languages such as Modelica, MetaModelica and C.

The template language has been designed to be strongly typed and efficient. It is compiled into MetaModelica and not interpreted as many other text template languages. This makes it very efficient.

Similar to Stringtemplate, Susan is designed to follow the model-view-controller concept, and to be a simple functional-style language.

- model – the intermediate tree (AST), to be converted to text according to the view.
- view – the mapping to text provided by the template functions.
- controller – the (sometimes conditional) traversal of the tree to create a view from the model.

The value of this principle is strongly argued in [17], according to experience with the ST functional template language [18] in the StringTemplate system. Such separation gives more flexibility (multiple views), easier maintainability, better reuse, more ease-of-use, etc.

It is argued that the template language should be kept simple, program computation logic should not be too much intertwined with emitting text. If complex computation needs to be done, it should instead be done on the model (in our case the AST).

Therefore, the Susan language has been designed be simple, only provide a mapping to text, and simple conditional tests and pattern matching. If more complicated computations should be done, it should be done on the model data structure (the tree), before transferring it to the template text output phase.

There are many template languages intended for C and Java users, but Susan is currently the only text template language adapted for Modelica and MetaModelica users.

The Susan template language is strongly typed and compiled into MetaModelica. The current prototype C code generator written in the template language is only approximately 10% slower than the current handwritten code generator, and this figure can probably be improved with some tuning.

Chapter 2

Template Language Features

2.1 Preliminaries

2.1.1 Predefined Text Data Type

The template language supports a predefined `Text` data type. The implicit result type of all template functions is type `Text`. Values of type Text can be very efficiently converted to `String`, or may sometimes be `String`. Buffers passed as reference parameters are always of type `Text`.

2.2 Template Text-with-hole Constructors and Template Holes

A text template is a text with holes. Inside a hole there can be any valid template expression. For example, the following text has three holes. Inside a hole is is possible to have a general template expression, of which the most simple form is just a name. Such a template expression is evaluated and converted to text during the evaluation of the template.

```
This is text in the template. This is text in the template. This is text in the
template. <%Hole-templ-expr1%> This is text in the template. This is text in the
template. This is text in the template. This <%Hole-templ-expr2%>is text in the <%Hole-
templ-expr3%>template. This is text in the template.  This is text in the template. This
is text in the template. This is text in the template...
```

Template holes are started by `<%` and ended by `%>`, i.e., as in `<%name%>`. When `<%` is needed as text, its first character need to be ecaped by `\`, i.e., `\<%`. There are currently two forms of template text-with hole constructors:

- Single-quote `'  '` text-with-hole constructor. All characters are included verbatim as-is, except holes starting with `<%` and `'` which ends the text. Those can be included by prefixing with the escape code backslash as in \<% and `\'`.
- Multi-Line << >> text-with-hole constructor. The template text starts on the line after << and ends including the line before >>. All characters are included verbatim as-is, except holes starting with `<%` and >> which ends the text. Those can be included by prefixing with the escape code backslash as in `\<%` and `\<<`.

It is actually possible to have multiple lines also in the single-quote variant, but then line-counting, alignment and indentation options do not work. Such options are only supported by the Multi-line variant.

Example of the single-quote `'  '` text-with-hole constructor:

```
'Output text <%templ-expr%>.'
```

Example of the Multi-Line << >> text-with-hole constructor:

```
<<
Output text <%templ-expr%>.
>>
```

Example within a template function

```
template functionInput(ModelInfo modelInfo) ::=
match modelInfo
case MODELINFO(vars=SIMVARS) then
<<
int input_function()
{
  <% (vars.inputVars |> SIMVAR(__)hasindex index0  =>
    '<%cref(name)%> = localData->inputVars[<%index0%>];') ;separator="\n"%>
  return 0;
}
>>
end match
end functionInput;
```

2.3 Template Expressions

Template expressions can consist of the following:

- Simple names, see Section 2.3.2.
- Quoted names, see Section 2.3.2.
- Double-quoted string constants, see Section 2.3.3.
- Single-quote text-with-hole constructors, see Section 2.2.
- Multi-line text-with-hole constructors, see Section 2.2.
- Conditional expressions, see Section 2.3.6.
- Match-expressions, see Section 2.5.
- Let-expressions, see Section 2.7.
- Function calls, see Section 2.4.3.
- Iterator expressions, see Section 2.6.
- Parenthesized expressions, see Section 2.3.4.
- Vector/list-constructors, see Section 2.3.7.
- Option expressions, see Section 2.8.

The full grammar of template expressions is as follows:

- ?? fill in

2.3.1 Automatic Conversion to String Data

Data retrieved from bound variables or returned from called MetaModelica functions in template expressions is automatically converted to string values according to the following.

Any auto-to-string-convertible bound value can be used.

- Automatic to-string conversion applies to the elementary types: `String`, `Integer`, `Real`, `Boolean`.
- Values of `Option` type are output for `SOME` values when the option type is auto-to-string-convertible (recursively).
- values of type `list` and `Array` are concatenated when element type is auto-to-string-convertible.

2.3.2 Bound Variable Reference, name or $'name'

A variable bound to a value is referenced by using its name, e.g. `valueName`, within a template expression (but not in the actual verbatim template text). Syntax:

```
valueName
```

An alternative dollar-quoted variant is similar to but different from Modelica's single-quoted identifiers:

```
$'valueName'
```

The `$'valueName'` means the same identifier as `valueName`, whereas the Modelica `'valueName'` includes the single-quotes in the identifier.

The value of a referenced bound variable name is retrieved and automatically converted to text according to Section 2.3.1. Any auto-to-string-convertible bound value can be used..

Examples where dollar sign or space or + is in the identifier, or just an ordinary name:

```
$'quoted id+23121'

$'$'

ordinaryName
```

Examples when the keyword `let` needs to be an identifier, e.g. as in these examples:

```
field.$'let'

case RECORD($'let'=FOO) then ...
```

2.3.3 Verbatim String Constants

Double-quoted string constants, e.g. `"string constant"`, are available and exactly respect Modelica stirng semantics, which is defined as follows:

```
STRING = """ { S-CHAR | S-ESCAPE } """
S-CHAR =  any member of the Unicode character set (http://www.unicode.org; but use UTF-8
          for storing on files) except double-quote """, and backslash "\"
```

Escape codes for certain control characters can be given in strings as follows and are defined in the same way as the C99 standard:

```
S-ESCAPE = "\'" | "\"" | "\?" | "\\" |
           "\a" | "\b" | "\f" | "\n" | "\r" | "\t" | "\v"
```

2.3.4 Parentheses

Parentheses are allowed, and has the normal interpretation used in almost all programming langaugs, i.e. giving priority in the order of evaluation. Expressions in innermost parenthesis will be evaluated first.

2.3.5 Reduction Expressions

If reduction to text is desired, you should use: '<< `expression-to-reduce` >>', which will reduce it to a single text item. This is usually not needed, since eventually the whole template expression including its parts will be reduced to text before being output.

2.3.6 Conditional Expressions

The conditional expression evaluates *templ-exp*$_1$ as the result of the expression when the condition is satisfied, otherwise value of *templ-exp*$_2$ is the result. When the **else** branch is not specified, an empty string is implied.

Syntax:

if [**not**]$_{opt}$ *condition* **then** *templ-exp*$_1$
[**else** *templ-exp*$_2$]$_{opt}$

The `condition` is intended to test values for their non-zero/zero-like values. Values of these types are allowed with the following semantics. Zero-like values are treated as false in the Boolean sense:

```
Boolean                  true     / false
```

```
Integer or Real         non-0      / 0
String                  non-empty  / ""
list or Array           non-empty  / { } empty list/array
Option                  SOME       / NONE
```

However, testing for the result of a template function (returning result of type `Text`) is not allowed:

```
if templ() then ...    // Error, cannot test conditionally on template function!
```

Example:

```
template globalDataVarNamesArray(String name, list<SimVar> items) ::=
if items then
<<
char* <%name%>[<%listLength(items)%>] = {<% (items |> SIMVAR(__) =>
  '"<%crefSubscript(origName)%>"' ) ;separator=", "%>};
>>
else
<<
char* <%name%>[1] = {""};
>>;
end globalDataVarNamesArray;
```

2.3.7 Vector/list Constructor {}

A vector/list constructor {} is useful in conjunction with conditional insertion of separators.

The separator is only inserted if there are two or more non-empty expressions (i.e., expressions not resulting in empty strings).

```
{ expr1, expr2 } ;separator = ","
```

This can be used for construction of a general list with many elements, as in the following:

```
{ expr1, expr2, expr3, ... exprN } ;separator = ","
```

Example using {}:

```
<<
char var_attr[NX+NY+NP] = {
    <% { (vars.stateVars |> SIMVAR(__) =>
        '<%globalDataAttrInt(type_)%>+<%globalDataDiscAttrInt(isDiscrete)%> /*
<%cref(origName)%> */'
      ",\n"),
      (vars.algVars of SIMVAR(__) =>
        '<globalDataAttrInt(type_)%>+<%globalDataDiscAttrInt(isDiscrete)%> /*
<%cref(origName)%> */'
      ",\n"),
      (vars.paramVars of SIMVAR(__)=>
        '<%globalDataAttrInt(type_)%>+<%globalDataDiscAttrInt(isDiscrete)%> /*
<%cref(origName)%> */'
       ",\n") }
    ;separator=",\n"%>
```

2.4 Template Function Declaration and Call

2.4.1 Template Function Declaration

A template function is declared as follows:

```
template funcname(Argtype1 arg1, Argtype2, arg2, ...) "Optcomment" ::=  templatebody
end funcname;
```

The implicit result type of the template function is `Text`, which always holds and need not be specified.

Example:

```
template functionInput(ModelInfo modelInfo) ::=
  match modelInfo
  case MODELINFO(vars=SIMVARS) then
<<
int input_function()
{
  <% (vars.inputVars |> SIMVAR(__) hasindex index0 =>
    '<%cref(name)%> = localData->in putVars[<%index0%>];') ;separator="\n"%>
  return 0;
}
>>
  end match
end functionInput;
```

2.4.2 Declaring External Imported MetaModelica Functions

A MetaModelica function that is to be called from within a template expression may have at most one output result and needs to be declared in an interface package, see Section 2.9.

The signature of the function is specified in MetaModelica syntax for the package it belongs to.

For example, the MetaModelica function `crefSubIsScalar` below is specified as an external function that can be imported from package `SimCode`.

```
interface package MyInterface

package SimCode

  function crefSubIsScalar
    input DAE.ComponentRef cref;
    output Boolean isScalar;
  end crefSubIsScalar;

end SimCode;

end MyInterface;
```

2.4.3 Template Function Call

Template functions are called in the same way as functions in general, e.g.:

```
fname(arg1, arg2, ... argN);
```

Buffer arguments passed to reference formal parameters need to be prefixed by & in the call. The `Text` result of the template evaluated with the actual parameters is the output of the template function call. Formal parameters are strongly typed. Automatic to-string conversion of actual parameters applies when appropriate.

2.4.4 Call of Imported External MetaModelica or C Functions

It is possible to call external MetaModelica functions from template functions. Such external functions need to be declared (Section 2.4.2) in an interface package (Section 2.9) which must be imported through an import statement into the package where the external functions are called.

You can use the return value of an external function natively with its original type, have the return value automatically converted to string, or call functions that return no value. Only external functions which return zero or one arguments can be used. Examples of these three cases follows.

2.4.4.1 Using return value natively with its original type

The return value from a called function can be used as is without being converted to text. For example:

```
template crefSubIsScalarHuman(DAE.ComponentRef cref) ::=
  if crefSubIsScalar(cref) then
    "this cref has scalar subscript"
  else
    "this cref does not have scalar subscript"
end crefSubIsScalarHuman;
```

Here the return value from the external imported function `crefSubIsScalar` is a boolean which is used as a boolean in the if-expression. The return value could also have been stored in a variable `resBool` as follows:

```
template crefSubIsScalarHuman(DAE.ComponentRef cref) ::=
  let resBool = crefSubIsScalar(cref)
  if resBool then
    "this cref has scalar subscript"
  else
    "this cref does not have scalar subscript"
end crefSubIsScalarHuman;
```

Alternatively the return value from `crefSubIsScalar` could be passed immediately to another template function `crefSubIsScalarHumanFromBool` like this:

```
template crefSubIsScalarHuman(DAE.ComponentRef cref) ::=
  crefSubIsScalarHumanFromBool(crefSubIsScalar(cref))
end crefSubIsScalarHuman;

template crefSubIsScalarHumanFromBool(Boolean resBool) ::=
  if resBool then
    "this cref has scalar subscript"
  else
    "this cref does not have scalar subscript"
end crefSubIsScalarHumanFromBool;
```

The return value could be used natively in other contexts as well where it makes sense.

2.4.4.2 Call with return value converted to string

The `tmpTick` external function called below returns an integer that is automatically converted to a string when used in a template expression.

```
template uniqueName() ::=
  let uniqName = 'tmp<%System.tmpTick()&>'
  uniqName
end uniqueName;
```

Or just like the following:

```
template uniqueName() ::=
  'tmp<%System.tmpTick()%>'
end uniqueName;
```

2.4.4.3 Call with empty return value

It is possible to call an external function which has no return value within the following variant of let-expression:

```
let () = noFuncRet() body-expr
```

For example,

```
template writeSomeTextToFile() ::=
  let content = "some text"
  let () = textFile(content, "file_name.txt")  // Call with empty result, e.g. to create file.
  () // This template function returns an empty result, e.g. as void in C, only its side effect is used.
end writeSomeTextToFile;
```

2.4.5 Declaring Reference (buffer) Formal Parameters in a Template Function

In a template function header, & is used before the reference parameter name, and the type in such cases is always `Text`.

Example:

```
template funcname(Argtype1 arg1, Text &arg2Reference) ::= ...
```

2.4.6 Using Reference (buffer) Formal Parameters in a Template Function

A reference parameter formal parameter always must be prefixed by & when it is used, e.g. when when passing it to another function or updating it in a `let ...` += text append statement (Section 2.7.3).

```
& buffername
```

In the template function `functionBody` we declare a buffer `varDecls` that can be updated. This buffer is passed to the template function `tempDecl`, prefixed with & in the call.

Further down, within the template function `tempDecl`, a side effect is performed on the reference parameter `varDecls`, it is updated, i.e., appended to by the `&varDecls +=` operation.

```
template functionBody(Function fn) ::=
  match fn
   case FUNCTION(__) then
      ...
      let &varDecls = buffer ""
      let retVar = tempDecl("modelica_real", &varDecls)  // inserted & here
      ...
      <<
      ..
      {
        ...
        <%varDecls%>
      }
      >>
  end match
end functionBody;

template tempDecl(String ty, Text &varDecls) ::=
  let newVar = 'tmp<%System.tmpTick()%>'
  let &varDecls += '<%ty%> <%newVar%>; <%\n%>'   // varDecls is updated, appended
  newVar  // return newVar
end tempDecl;
```

2.4.7 Allowed Side-Effects in Template Functions

The only direct side-effects allowed in template functions are append (+=)operations on buffer variables. Side effects can also occur through calls to external functions.

2.5 Match Expressions and the Idea of Pattern Matching

The match expression in the template language. is mostly used for discrimination of tree structure nodes (abstract syntax) of some MetaModelica uniontype. First we give a brief introduction to union types and the idea of pattern matching.

2.5.1 The MetaModelica uniontype Construct

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

- A union type specifies a union of one or more record types.
- Union types can be recursive – they can reference themselves.

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

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

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

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

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

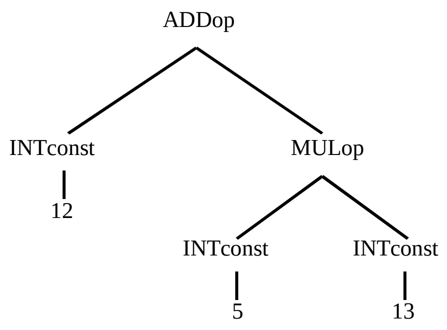

Figure 2-1. Abstract syntax tree of 12+5*13 in the language Exp.

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

All union type referenced in the template functions need to be declared in an interface package, see Section 2.9.

2.5.2 Match Expressions and Pattern Matching

The match expression is mostly used for discriminating of tree structure nodes of union types. The syntax is as follows, where an optional `else` can appear last in the list of cases.

```
match value-exp
  case pattern-exp1 then templ-exp1
  case pattern-exp2 then templ-exp2
  ...
  [else pattern-expn then templ-expn]opt
[end match]opt
```

The `templ-exp`$_i$ template expression of the first case that matches the `value-exp` against its `pattern-exp`$_i$ is evaluated as the result of the whole expression.

- The value computed by the expression after the `match` keyword is matched against each of the patterns after the `case` keywords in order; if one match fails the next is tried until there are no more case-branches in which case (if present) the else-branch is executed.
- When no pattern matches, the result is empty string.
- When a pattern matches, all pattern variables in the pattern become bound to corresponding parts in the structured value `value-exp`. Also, the scope of the top-most record constructor, if present in the pattern, is opened to make all its fields available as read-only variables, see Section 2.5.6.

2.5.2.1 Simple Pattern Matching

A very simple example of a match-expression is the following code fragment, which returns a number corresponding to a given input string. The pattern matching is very simple – just compare the string value of `s` with one of the constant pattern strings `"one"`, `"two"` or `"three"`, and if none of these matches return 0 since the wildcard pattern _ (underscore) matches anything.

```
match s
  case "one"   then 1
  case "two"   then 2
  case "three" then 3
  case _       then 0
end match
```

Using an else-branch we could instead have written:

```
match s
  case "one"   then 1
  case "two"   then 2
  case "three" then 3
  else   0
end match
```

The following is an example of pattern matching against a tree structure as in Figure 2-1, e.g.

```
match inExp
  case INTconst(int=v1) then ...
  case ADDop(exp1=e1,exp2=e2) then ...
  case SUBop(__) then ...
  case MULop(__) then ...
  case DIVop(__) then ...
  case NEGop(__) then ...
end match
```

The first case uses named pattern pattern matching (Section 2.5.4 where the single named field of the `INTconst` record is `int`, and the pattern variable is `v1`:

```
    case INTconst(int=v1) then ...
```

The interpretation is to match the `inExp` value against the special case pattern `INTconst(int=v1)` If there is a match, the pattern variable `v1` will be bound to the corresponding part of the tree. For the left subtree in Figure 2-1, `INTCONST(12)`, `v1` will be bound to 12.

We now turn to the second rule, which has a named pattern for the `ADDop` node:

```
    case ADDop(exp1=e1,exp2=e2) then ...
```

For this case to apply, the pattern `ADDop(exp1=e1,exp2=e2)` must match the actual `inExp` value, which is an abstract syntax tree. If there is a match, the variables `e1` and `e2` will be bound the two child nodes of the `ADDop` node, respectively, visible in Figure 2-1.

A more convenient way of matching is to use the implicit scope opening (Section 2.5.6) of patterns that match. Instead of specifying a named pattern with a number of field names and pattern variables, one can use the following form that matches any ADDop node:

```
    case ADDop(__) then ...
```

The scope of the `ADDop` node is automatically opened (Section 2.5.6), to make the fields `exp1` and `exp2` available in the current scope bound to the corresponding subtrees of the `inExp` value. Thus, it is unnecessary to introduce the pattern variables `e1` and `e2`. Instead we can use `exp1` and `exp2` directly to refer to the subtrees of the `ADDop` node.

2.5.2.2 Using Pattern Matching in Template Functions

In the following example the template function `exp` is calling itself recursively. The scope of the `ICONST` constructor is automatically opened to make its field called `value` available, and the `PLUS` constructor is opened automatically by using the `PLUS(__)` pattern, see Section 2.5.4 2.5.6.

```
template exp(Exp ep) ::=
  match ep
  case ICONST(__) then value
  case PLUS(__)   then '(<%exp(lhs)%> + <%exp(rhs)%>)'  // Open scope, see Section 2.5.6
end exp;
```

where the type `Exp` is:

```
uniontype Exp
  record ICONST
    Integer value;
  end ICONST;
  record PLUS
    Exp lhs; Exp rhs;
  end PLUS;
end Exp;
```

The `end match` is usually optional, but mandatory in case of nested match expressions as below:

```
match a
 case RECORD1(__) then
  match b
   case RECORD2(__) then expr
  end match
 case ...
```

The above template function exp with an optional `end match` added:

```
template exp(Exp ep) ::=
  match ep
    case ICONST(__) then value
    case PLUS(__)   then '(<%exp(lhs)%> + <%exp(rhs)%>)'  // Open scope, see Section 2.5.6
  end match
end exp;
```

2.5.3 Pattern Expressions in General

Forms of patterns allowed:

- Constant value, e.g. `1`, `"foo"`, `3.14`, etc.
- Constructor with parenthesis and double underscore, e.g. `RELATION(__)`in the example below, with implicit opened record scope, see Section 2.5.6.
- Constructor with parenthesis and named pattern variables, `CONSTRUCTOR(name1=pat1, name2=pat2, ...)` e.g. the `CALL(....)` pattern in the example below.
- A tuple constructor `(arg1, arg2, arg3, ...)` using parentheses.
- A list constructor `{arg1, arg2, arg3, ...}` using curly braces.
- An underscore _ which matches anything.
- A single identifier, `name`, acts as a pattern that can be bound to anything.
- An as-expression, e.g. `name as pattern-expr`, Section 2.5.5.

Example:

```
template zeroCrossingTpl(Integer index, Exp relation, Text &varDecls) ::=
match relation
```

```
case RELATION(__) then
  let preExp = ""
  let e1 = daeExp(exp1, contextOther, preExp, &varDecls)
  let op = zeroCrossingOpFunc(operator)
  let e2 = daeExp(exp2, contextOther, preExp, &varDecls)
  <<
  <%preExp%>
  ZEROCROSSING(<%index%>, <%op%>(<%e1%>, <%e2%>));
  >>;
case CALL(path=IDENT(name="sample"), expLst={start, interval}) then
  let preExp = ""
  let e1 = daeExp(start, contextOther, preExp, &varDecls)
  let e2 = daeExp(interval, contextOther, preExp, &varDecls)
  <<
  <%preExp%>
  ZEROCROSSING(<%index%>, Sample(*t, <%e1%>, <%e2%>));
  >>;
case _ then
  <<
  ZERO CROSSING ERROR
  >>
end zeroCrossingTpl;
```

2.5.4 Record Constructor Pattern Expressions

The following two forms are possible:

- Constructor with parenthesis and double underscore, e.g. `RELATION(__)`in the example below, with implicit opened record scope, see Section 2.5.6.
- Constructor with parenthesis and named pattern variables, `CONSTRUCTOR(name1=pat1, name2=pat2, ...)`. (??Question: does implicit scope opening also apply for this kind of pattern?)

Some forms of patterns:

```
REC(field = ASUB())   // constructor with no fields
REC(field = ASUB(field1=_))  // constructor with one field named field1
REC(field = ASUB(__))  // constructor with any number of fields
REC(field = ASUBB)  (mis-spelling of ASUB, then becomes a pattern variable ASUBB
```

2.5.5 Pattern Expression Variable Binding Using as

The following pattern matches `pattern-expression` and binds the variable `var` to the matched value if the match is successful

```
var as pattern-expression
```

This construct is the same in MetaModelica and Susan, and essentially the same as what is found in several functional language.:

2.5.6 Implicit Opening of Record Constructor Scopes in Patterns

A record constructor pattern in a match-expression case-rule opens the scope of the record and make all the record fields available (read-only). (This is similar to instanceof in Java)

For example in the `ASSIGN(__)` pattern below, the record fields `lhs` and `rhs` becomes available. In the `WHILE(__)` pattern below, the condition and statements fields become available.

Only the scope of the outermost constructor in a nested constructor pattern is opened.

Example:

```
template statement(Statement stmt) ::=
  match stmt
  case ASSIGN(__) then <<
<%exp(lhs)%> = <%exp(rhs)%>;
  >>
  case WHILE(__)  then <<
while(<%exp(condition)%>) {        // call to template function exp
  <%statements |> st => statement(st) ;separator="\n"%>
}
  >>;
end statement;
```

for

```
uniontype Statement
  record ASSIGN
    Exp lhs; Exp rhs;
  end ASSIGN;
  record WHILE
    Exp condition;
    list<Statement> statements;
  end WHILE;
end Statement;
```

Only the scope for the outermost constructor `REC1`, is opened in a nested pattern expression. Use `as`-notation for the nested ones.

Example, with both `REC1` and `REC2` containing `field1`:

```
REC1(...  x as REC2(...) ...)
```

Use of `field1` will denote `field1` within `REC1`, and `x.field1` will denote `field1` within `REC2`.

2.6 Iterator Expressions

Iterator expressions behave similarly to array or list comprehensions in functional languages, but has the arguments in a different order. The following simple variant is a trinary operator that iterates elem over the elements in the element-list and constructs a new list from the template-expressions.

```
element-list |> elem => template-expression
```

This is equivalent to a Modelica iterator expression (also called array- or list-comprehension):

```
{template-expression for elem in element-list}
```

For example, the following iterator expression:

```
{"a", "b", "c"} |> x => 'U<%x%>!')
```

would produce the following list of items:

```
{"Ua!", "Ub!", "Uc!"}
```

that is eventually reduced and concatenated to a single text string:

```
"Ua!Ub!Uc!"
```

The general form of the operator allows a general pattern to iterate over and match the elements in `element-list`. Only the elements that match `elem-pattrn` will be forwarded and used to construct instances of `template-expression`. This is typically used for filtering applications.

```
element-list |> elem-pattrn => template-expression
```

The operator has the following properties:

- It is a trinary operator with three operands. It also has an optional form as a quad operator with four operands when the `hasindex` keyword is present, see Section 2.6.1.
- It has an expression as its third argument, not a function as with the related map function.

- Being a non-associative operator forces the sues parentheses for more clear readability since the evaluation order is always visible from the syntax.
- The left-most argument `elements` must be a list or an array.
- The left-to-right property of the operator can be used to write a series of such operations in a left-to-right fashion – the results of an iterator expression to the left can immediately be fed into an iterator expression to the right – sometimes known as piping.

The iterator expression is often used together with a separator and other options applicable for multi-result values like lists.

Example 1.

```
template gentlemen(list<String> names) ::= <<
Hello <%(names |> name => 'Mr. <%name%>') ;separator=", "%>!
>>
end gentlemen;
```

The output of the template function call `gentlemen({Adam, Eric, Carl})` will be:

```
Hello Mr Adam, Mr Eric, Mr Carl!
```

```
template pairList(list<tuple<String,Integer>> pairs) ::=
<<
Pairs: <%pairs |> (s,i) => '(<%i%>,<%s%>)' \n ;anchor%>.
>>
end pairList;
```

This filters out only values of the record type `ICONST`, see also the match expression Section 2.5.

```
template intConstantsList(list<Exp> expLst) ::=
  (expLst |> ICONST(__) => value ;separator=", ")
end intConstantsList;
```

An example where a list of variable declarations is generated:

```
<%variables |> var as VARIABLE(__) => '<%varType(var)%> <%cref(var.name)%>;'
 ;separator="\n"%>
```

Example:

```
let removedPart = (removedEquations |>  eq  =>
  '<%equation_(eq, contextSimulationNonDiscrete, varDecls)%>' ;separator="\n")
```

Example:

```
(list1 |> x
  => 'I love <%x%> do') |> y   => 'Dumb <%y%>;'
```

More examples:

Ex 1:

```
<%zeroCrossingsNeedSave |> vars => (
  <<
  case <%vars.index0%>:
    <%vars |> SIMVAR(__)=>'save(<%cref(name)%>);' ;separator="\n"%>
    break;
  >> )
;separator ="\n"%>
```

Ex 2: The `includes` binding is from opening the `EXTERNAL_FUNCTION` constructor scope.

```
  <%functions |> EXTERNAL_FUNCTION(__)=>
    (includes ;separator= "\n") ;separator="\n"%>
```

2.6.1 Iterator Expressions with Iteration Index Values

There is a quad operand version of the iterator expression operator with the optional **`hasindex`** keyword (and the further optional `fromindex` keyword), followed by an identifier, i.e., a total of four (or five) operands:

```
elements |> elem-pattrn [hasindex myindex0 [fromindex startindex ] ] => template-
expression
```

The variable after `hasindex` gives the ordinal number of the corresponding element in elements starting counting from 0 as default, but it is possible to start from another number, e.g., 1, by using `fromindex 1`.

The following examples use the `hasindex` keyword. One example is using the optional `fromindex` keyword to specify the start index offset to specify non-zero start indexes such as having 1 as the lowest index (as in Modelica):

```
<multi-val-expr |> el hasindex myindex0 => templ(el, myindex0) ;options>
<multi-val-expr |> el hasindex myindex1 fromindex 1 => templ(el, myindex1)>
```

2.7 Let Expressions with Name Bindings and Text Buffers

2.7.1 let Binding of Local Named Text Values

The language allows local definitions that can be referred to in the template body. A local definition `name` is bound to a text value `expr` and accessible within the scope of `bodyexpr`:

```
let name = expr  bodyexpr
```

Several let binding can be nested:

```
let name1 = expr1
let name2 = expr2
let name3 = expr3
bodyexpr
```

This is equivalent to:

```
let name1 = expr1
(let name2 = expr2
(let name3 = expr3
bodyexpr ) )
```

since the let-operator is left associative.

Example with several uses of let:

```
template functionDaeOutput2(list<SimEqSystem> nonStateDiscEquations,
                   list<SimEqSystem> removedEquations) ::=
  let &varDecls = buffer ""
  let nonSateDiscPart = (nonStateDiscEquations |> eq =>
           '<%equation_(eq, contextSimulationDescrete, varDecls)%>' ;separator="\n")
  let removedPart = (removedEquations |> eq =>
           '<%equation_(eq, contextSimulationDescrete, varDecls)%>' ;separator="\n")
<<
/* for discrete time variables */
int functionDAE_output2()
{
  state mem_state;
  <%varDecls%>

  mem_state = get_memory_state();
  <%nonSateDiscPart%>
  <%removedPart%>
  restore_memory_state(mem_state);

  return 0;
}
```

```
>>
end functionDaeOutput2;
```

2.7.2 let Binding of Buffer Variables and their Use

A buffer variable can be introduced in the following way through a let-binding:

```
let &name = buffer text-expression  body-expression
```

The reference, name, is immutable, but the contents is mutable since it is a buffer. The value of the buffer is initialized to the `text-expression`. As usual, the let-binding of name can be accessed within the scope of the body-expression.

The following language rules apply regarding text buffers

- A text buffer can only be appended to by the += operator in a let expression (Section 2.7.3).
- A text buffer can be passed as an argument to a template function by marking it with & at the call. This is a reference parameter that can be modified by the called function, i.e., an in-out parameter (Section 2.4.6)
- A text buffer can only be referenced inside text template expressions in the template function it is declared in. (text buffers can only be appended to in template functions they are passed to)

2.7.3 Appending a String to a buffer Variable

It is possible to perform a side effect of appending a string `expr` at the end of a buffer variable `var1`:

```
let &var1 =+ expr
```

This is equivalent to the following Modelica code:

```
var1 := var1 + expr;
```

Example 1:

```
let &preExp += 'create_index_spec(&<%tmp%>, <%nridx_str%>, <%idx_str%>);<%\n%>'
```

Example 2:

```
template tempDecl(String ty, Text &varDecls) ::=
  let newVar = 'tmp<%System.tmpTick()%>'
  let &varDecls += '<%ty%> <%newVar%>; <%\n%>'   // varDecls is updated, appended
  newVar  // return newVar
end tempDecl;
```

2.7.4 Reference (buffer) Formal Parameters in Template Functions

In a template function header, & is used before the reference parameter name, and the type in such cases is always Text.

Example:

```
template funcname(Argtype1 arg1, Text &arg2Reference) ::= ...
```

2.8 Formatting, Separator, and Indentation Options

A number of options can be specified with the option operator to control formatting and indentation of a template expression templ-exp.

templ-exp ;opt_1=val_1 ;opt_2 ;opt_3=val_3 ... ;opt_n=val_n

It is usually used inside a hole, and can optionally be enclosed in parentheses as any other expression, e.g.:.

(templ-exp ;opt_1=val_1 ;opt_2; ... ;opt_n=val_n)

This is a semi-colon 2n+1-ary left associative operator. All options and option-values are collected, and simultaneously applied to the leftmost operand, the `templ-exp`.

2.8.1 Indentation controlling options

- `anchor` – relative to start of the expression
- `absIndent` – absolutely from the line beginning
- `relIndent` – relative to the actual indent
- `indent` – immediate indent and then relative to the actual indent

2.8.2 Multi-Value Formatting Options

It is *important* to note that these options are only valid for values that contain multiple values, e.g. lists/arrays.

- `separator` – separator text inserted between the results in the multiple-value list/array.
- `align` – the number of results to be aligned by `alignSeparator` – separator for aligning `alignOffset` – start align counting offset
- `wrap` – number of characters to be wraped by `wrapSeparator` – separator for wrapping
- `empty` – value substituted for empty results
- `countEmpty` – *To be implemented* **-** count empty results when using `hasindex` in an iterator expression
- `separateEmpty` – *To be implemented* – used together with `separator` to control separation of empty results. Default is false.

2.8.3 Options

Expression options can be specified only in the direct lexical context of **<** … **>** or (…).

Indentation controlling options control indentation that occurs *before outputting the first non-space character after a new line* inside of the option affected output text.

All indentation options are of type integer where usage without ‘=’ defaults to 0.

The `indent` option outputs the specified number of spaces immediately and then behaves like `relIndent`.

Multi-values formatting options can be applied for all expressions that (possibly) results in concatenation of multiple results, i.e., list or array values, map expressions and (pseudo)list construction expressions.

The `separator` option is used for inserting a separator string between elements of multiple-value expressions t. Default separator value is empty string.

The `align` and `wrap` options are of type integer where a positive value means a number of results or characters, respectively, after which a value of the `alignSeparator` or `wrapSeparator` will be output. Default values of `alignSeparator` and `wrapSeparator` is new-line character. Default value of `align` option is `10` and the default of `wrap` option is `100` (these values are used when the option is specified without a value – i.e., not using ‘=’).

The `alignOffset` option can be used to set the start counting offset from whichthe `align` option counts its effect. Default is 0.

The `empty` option is of string type (more precisely, `Option<String>`), and when specified, its value is used whenever the concatenated result is an empty string. This option has impact on the counting and separation semantics (which is not intuitive in several scenarios). By default, this option is “unset” (internally `NONE`) that means that there is no replacement for empty results and the separation and index counting by default DOES NOT count/separate empty results. When this option is used without a value (without ‘=’ after it), it is equivalent to empty="" (empty string) and it means that "there is a value" for every result, so the counting and separation takes the value as non-empty (it is counted/separated).

To address this complex semantics (a mess), the two options `countEmpty` and `separateEmpty` will be implemented to make the semantics much clearer and explicit. Particularly, this option will only dictate the replacement value for empty result and it will have by default the value of an empty string (not `NONE` as now).

The `countEmpty` – *To be implemented* - option is of Boolean type with default value true (in the current implementation it is false). When set to true, it means that an index variable defined by `hasindex` for this multi-value expression is advanced for empty results, otherwise not. It is important to note that the emptiness is checked against the actual result value regardless of the `empty` option value (so the `empty` option is used effectively only for replacement of the empty results while not affecting the emptiness checks).

See also Section 2.6.1, for more information about `countEmpty` and `hasindex`.

Examples:

```
//note the automatic indentation by 2 spaces
template lines2(list<String> lines) ::= <<
  <%lines ;separator=\n%>
>>
end lines2;


//align by 15 values and anchor the output 1 space after {
template intArr(list<Integer> values) ::= <<
int[] myArr = { <%values ;separator=", " ;align=8 ;anchor%> };
>>
end intArr;


/* example output:
int[] myArr = { 1, 2, 3, 4, 5, 6, 7, 8,
                9, 10, 11, 12, 13, 15, 16,
                17, 18, 19, 20 };
*/
```

2.9 Interface Packages

Template functions in the Susan language are grouped in *packages*, currently with file extension `.tpl`. Each template package can import one or more interface packages, i.e., that defines sets of AST type definitions and/or function signatures. Each interface packages uses MetaModelica syntax and resides in a separate `.mo` file.

An interface packages has the same properties as an ordinary package, but with the following differences:

- Imported items are re-exported the naming they are given depending on the import.
- Imported packages are re-exported with a restricted view – only the items explicitly declared are re-expred.
- Functions in the interface package can only be function headers for giving the function signatures.
- Union types which are re-exported only make the explicitly declared constructors and fields visible in the exported view, i.e., a so-called `type view` for that union type.
- How should an interface package be referred to in other packages (and in the .tpl file?) Answer: with an import statement, e.g. `import interface SimCodeInterface`. Multiple interfaces packages can be imported.

Note that the `public` and `protected` keywords should not be used in an interface package before the uniontype and function definitions.

Here we will show an example interface package that models the while loop example from the Modelica'2009 text template paper, and defines type views for the `Statement`, `Exp`, and `Operator` union types.

```
interface package InterfacePackageName
...
```

```
package OriginalPackageName

uniontype Statement  "Algorithmic stmts"
  record ASSIGN  "An assignment stmt"
    Exp lhs; Exp rhs;
  end ASSIGN;

  record WHILE  "A while statement"
    Exp condition;
    list<Statement> statements;
  end WHILE;
end Statement;

uniontype Exp  "Expression nodes"
  record ICONST  "Integer constant value"
    Integer value;
  end ICONST;

  record VARIABLE "Variable reference"
    String name;
  end VARIABLE;

  record BINARY  "Binary ops"
    Exp lhs; Operator op;  Exp rhs;
  end BINARY;
end Exp;

uniontype Operator
  record PLUS end PLUS;
  record TIMES end TIMES;
  record LESS end LESS;
end Operator;

end OriginalPackageName;

...
end InterfacePackageName;
```

The `OriginalPackageName` is the name of the original MetaModelica package where types corresponding to the type views in the interface package are fully defined. An interface package can use types from several packages. It usually specifies a subset of the original types defined in several packages and from these types suitable parts can be selected. For example, there can be additional union tags in the `Statement` type, but only those specified in the type view in the interface package can be used by templates that use this view. Similarly, more record fields can be originally defined in the `ASSIGN` record but only `lhs` and `rhs` can be read inside the template package with the view imported.

Interface package files with AST type views can be shared across different target languages as a kind of type interface to the compiler generated output ASTs (e.g., simulation code ASTs). It is also an essential feature to support scenarios where users are not allowed to see all original types (e.g., a commercial Modelica compiler) but still can see and use the intended subset to extend the code generator.

In addition to type views in interface packages, templates automatically understand all MetaModelica built-in types: `String`, `Boolean`, `Integer`, `Real`, `list`, `Option`, `tuple`, and `Array` types.

You can import multiple interface packages.

Example:

```
interface package SimCodeInterface

...

package builtin

  function listLength "Return the length of the list"
    replaceable type TypeVar subtypeof Any;
    input list<TypeVar> lst;
    output Integer result;
  end listLength;

end builtin;


package SimCode

  function crefSubIsScalar
    input DAE.ComponentRef cref;
    output Boolean isScalar;
  end crefSubIsScalar;

  ...

  uniontype Context
    record SIMULATION
      Boolean genDiscrete;
    end SIMULATION;
    record OTHER
    end OTHER;
  end Context;

  constant Context contextSimulationNonDescrete;

  ...
  type Variables = list<Variable>;
  type Statements = list<Statement>;
  type VariableDeclarations = Variables;

  uniontype Variable
    record VARIABLE
      DAE.ComponentRef name;
      Type ty;
      Option<DAE.Exp> value;
      list<DAE.Exp> instDims;
    end VARIABLE;
  end Variable;

  uniontype Statement
    record ALGORITHM
       list<DAE.Statement> statementLst;
    end ALGORITHM;
  end Statement;
...


end SimCode;


package DAELow

  uniontype ZeroCrossing
    record ZERO_CROSSING
      DAE.Exp relation_;
    end ZERO_CROSSING;
  end ZeroCrossing;

...

end DAELow;

...

end SimCodeInterface;
```

2.9.1 Interface Package Import into a Template File

To import the above example interface package into a template file, use the following syntax:

```
import interface SimCodeInterface;
```

By default, all imported symbols (imported types, functions, constants) are also accessible without the need of a package qualification, although it is recommended to visually distinguish from a template call to use proper package names in front of imported functions calls, for example (an extract from the `SimcodeTV`)

```
interface package SimCodeTV
…
package System
…
  function tmpTick
    output Integer tickNo;
  end tmpTick;
…
end System;
…
end SimCodeTV;
```

It is recommended to call this function in template :

```
System.tmpTick()
```

Instead just (working)

```
tmpTick()
```

(*Undocumented feature* - One can force usage of qualified lookup of names of an imported package by the usage of the **`protected`** keyword in front of the imported package in the interface file, but this usage of the protected keyword is might be redesigned in the future; with ?forcequalified?)

2.10 Template File Import into Another Template File

Importing of a template package into another template package is possible with **`import`** statement.

There is *unqualified* and *qualified* import with the semantics similar to (or the same) as in the standard Modelica language. Syntax:

```
import TemplatePackage.*;  //unqualified import of all symbols in TemplatePackage
import TemplatePackage;    //fully qualified import of TemplatePackage
```

The unqualified import does not require package name paths to be used when referencing an imported symbol. It is still possible to use a (fully-)qualified name when needed, for example when a local symbol hides the imported one (local symbols take precedence over imported ones).

The (fully-)qualified import of a template package forces the usage of fully qualified but shorter paths to referenced imported symbols (now, effectively, only the package name).

2.11 Template Error Handling

Generally, templates should not report any errors as templates are intended only to give a *textual view* of an abstract and correct (already checked) simulation code (or another text view of an abstract representation).

In practice, templates during development are often incomplete with respect to the input structures and an (template-)error mechanism is useful to cover these situations.

A template file can import (via an interface package) two functions from `Tpl` package (Susan’s runtime) `Tpl.addSourceTemplateError()` and `Tpl.addTemplateError()` to be able to report an error from within templates. Their signatures are:

```
function addSourceTemplateError
  "Wraps call to Error.addSourceMessage() funtion with Error.TEMPLATE_ERROR and one
   MessageToken."
  input String inErrMsg;
  input Absyn.Info inInfo;
end addSourceTemplateError;

//for completeness; although the addSourceTemplateError() above is preferable
 function addTemplateError
  "Wraps call to Error.addMessage() function with Error.TEMPLATE_ERROR and one
     MessageToken."
   input String inErrMsg;
 end addTemplateError;
```

The errors reported with `Tpl.addSourceTemplateError()` and `Tpl.addTemplateError()` are checked in the top-level MetaModelica functions `Tpl.tplString()` and `Tpl.noRet()`. These functions are intended to be the only template invocation entry points from MetaModelica code and they will fail when an error was reported inside the processed templates.

In conjunction to these functions, to be able to also report the source location of a template error, a built-in function `sourceInfo()`has been introduced that (magically) returns contextual `Absyn.Info` structure based on the position in the template file where it is "invoked". The `sourceInfo()` can only be used as a direct parameter into a template/imported function. No let binding of its value is currently possible (it will be possible when the full semantics of the let will be implemented).

Then, user can create his/her own error reporting templates. For example for a C++ generator

```
 template error(Absyn.Info srcInfo, String errMessage)
  "Example source template error reporting template to be used together with the
   sourceInfo() magic function.
   Usage: error(sourceInfo(), <<message>>)"
 ::=
 let() = Tpl.addSourceTemplateError(errMessage, srcInfo)
 <<

 #error "<% Error.infoStr(srcInfo) %> <% errMessage %>"<%\n%>
 >>
 end error;
```

(note: the `Error.infoStr()` function must be imported via an interface package)

An example using the above template:

```
 template literalExpConstBoxedVal(Exp lit)
 ::=
   match lit
   case ICONST(__) then 'MMC_IMMEDIATE(MMC_TAGFIXNUM(<%integer%>))'
   case BCONST(__) then 'MMC_IMMEDIATE(MMC_TAGFIXNUM(<%bool%>))'
   //… original cut for brevity
   case lit as SHARED_LITERAL(__) then '_OMC_LIT<%lit.index%>'
   else error(sourceInfo(), 'literalExpConstBoxedVal failed: <%printExpStr(lit)%>')
 end literalExpConstBoxedVal;
```

Chapter 3

References

[1] ANTLR. http://www.antlr.org. Access Nov 2007.

[2] Apache Software Foundation. Velocity Users Guide, 2008.: http://velocity.apache.org/engine/releases/velocity-1.6.1/user-guide.html. Jan 2009.

[3] Uwe Assmann. *Invasive Software Composition*. ISBN 3540443851, 9783540443858, 334 pages. Springer Verlag, 2003.

[4] Martin Fowler: *Domain Specific Language* http://www.martinfowler.com/bliki/DomainSpecificLanguage.html.

[5] Martin Fowler. *Domain Specific Languages* http://martinfowler.com/dslwip/

[6] Peter Fritzson. *Towards a Distributed Programming Environment based on Incremental Compilation*. PhD thesis no 109, Linköping University, April 13, 1984.

[7] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe*, 44/45, Dec 2005. http://www.openmodelica.org

[8] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 940 pages, Wiley-IEEE Press, 2004.

[9] Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proc. of the 4th International Modelica Conference*, Hamburg, Germany, March 7-8, 2005.

[10] Peter Fritzson, Adrian Pop, Kristoffer Norling, and Mikael Blom. Comment- and Indentation Preserving Refactoring and Unparsing for Modelica. In *Proc. 6th Int. Modelica Conf. (Modelica'2008)*, Bielefeld, Germany, March.3-4, 2008.

[11] Peter Fritzson, Pavol Privitzer, Martin Sjölund, and Adrian Pop. Towards a Text Generation Template Language for Modelica. In *Proceedings of the 7th International Modelica Conference (Modelica'2009*), Como, Italy, September.20-22, 2009.

[12] Google. ctemplate, 2008. http://code.google.com /p/google-ctemplate/. Accessed 2009.

[13] Kenneth C. Louden. *Programming Languages, Principles and Practice*. ISBN 0-534-95341-7, Thomson Brooks/Cole, 2003.

[14] Modelica Association. *The Modelica Language Specification Version 3.0,* September 2007. http://www.modelica.org.

[15] Martin Mikelsons. Prettyprinting in an interactive programming environment. In *Proc. of ACM SIGPLAN SIGOA symposium on Text manipulatio*n. Portland, Oregon, 1981.

[16] Eclipse website. http://www.eclipse.org. Referenced Nov 2007.

[17] Terence Parr. Enforcing Strict Model-View Separation in Template Engines. http://www. stringtemplate .org,. May 2004. Accessed May 2009.

[18] Terence Parr. [DRAFT] A Functional Language For Generating Structured Text. http://www.stringtemplate.org. May 2006. Accessed May 2009.

[19] Terence Parr. StringTemplate documentation. http://www.stringtemplate.org. Access May 2009.

[20] Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proceedings of the 4th International Modelica Conference*, Hamburg, , March 7-8, 2005.

[21] Adrian Pop, Peter Fritzson, Andreas Remar, Elmir Jagudin, and David Akhvlediani. OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging. In *Proc 5th International Modelica Conf. (Modelica'2006)*, Vienna, Austria, Sept. 4-5, 2006.

[22] Adrian Pop. *Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages*. www.ep.liu.se. PhD Thesis No. 1183, June 5, 2008.

[23] Martin Sjölund. *Bidirectional External Function Interface Between Modelica/MetaModelica and Java*. Master Thesis. Linköping Univ, Aug. 2009.