Toma

Draft Suggestion for TMQL

Author: Rani Pinchuk, Space Applications Services

Table of Contents

1 General Features

1.1 Semicolon

Each Toma statement has to end with a semicolon.

Example:

```
select $topic where $topic='cpu';

$topic
-------
cpu
```

1.2 White Space

White space is used in order to identify tokens. Apart from that, white space characters and new-lines have no meaning and can thus be used to indent statements.

1.3 Case Sensitivity

Toma is case sensitive, although the reserved words of the language are case insensitive (like in SQL).

Examples:

```
select $topic where $topic='cpu';

$topic
-------
cpu

SELECT $topic WHERE $topic='cpu';

$topic
-------
cpu

select $topic where $topic='CPU';

$topic
-------
```

The last query does not have any results, because there is no topic with the uppercase word 'CPU' as one of its names.

1.4 Toma Strings

Toma strings are defined by single quotes (like in SQL). A backslash before a quote escapes that quote. Example:

```
'this is a quoted string'

'and that\'s also a quoted string'
```

1.5 Comments

Comments are written by using the hash sign (#). Any text following the hash sign, and until the end of the line, is ignored.

Example:

```
select $topic where $topic='cpu'; #this is a comment
```

1.6 Topic Literals

A topic literal is an expression that resolves to a topic. The following table details the available topic literals in Toma. TODO – the CTM syntax should be used here.

Expression	Resolves to	Example
i<quoted-string>	a topic with that item identifier	i'host-location'
n<quoted-string>	topics with that name	n'Processor'
v<quoted-string>	topics with that variant	v'CPU'
si<quoted-string>	a topic with that subject identifier	si'http://www.topicmaps.org/subjectIndicator xtm/1.0/core.xtm#scope'
sl<quoted-string>	a topic with that subject locator	sl'http://tmra.be'

1.7 Naked Identifier

A naked identifier is a string without quotes around it and can be used as a shortcut to the id topic literal. It can be located within typing brackets (see section 2.6.2 on page 22), following the scope operator (the `@` sign described in section 2.7 on page 23) or as a role (after or before the arrow in the association expression described in section 2.4.1 on page 16).

In the following example `oc(location)` is resolved by taking "location" as a naked identifier. Therefore, both expressions of the example are resolved to the occurrences whose type has the id "location".

```
select oc(location);  #equivalent to oc(i'location');
```

In the same way the following can be written:

```
select $topic.name@en;  # equivalent to $topic.name@(i'en')
```

Hence, by default, Toma uses topic ids to refer to topics. However, other topic literals provide a way to refer to topics using another reference than the topic id (which is often automatically generated and thereby unknown to the user).

1.8 Toma Variables

Toma allows defining variables. A variable is written as a dollar sign followed by any letter (upper or lower case) or an underscore and optionally followed by any alphanumeric or underscore characters:

```
$<variable_name>
```

Each variable has a type. The types of the variables can be:

- topic
- locator
- name
- variant
- occurrence
- association

Note that no special type is defined for scopes, types or member roles as the engine that implements Toma always reifies the scopes, types and member roles - and therefore they are topics. The type of the variable is determined by the syntax or the semantics of the statement. The default type of a variable is topic.

Examples:

```
$a
$topic
$person
$association
$this_is_a_toma_variable
$variable23
```

Often, a variable is used only once. In that case, we do not care about its value, nor about its name. The only thing we do care about is that it is different from any other variable in the statement. In those cases we can use the **anonymous variable $$**. The anonymous variable is a variable like any other variable, but without a name. If there are several anonymous variables in a statement, they are all interpreted as different variables.

2 Toma Path Expressions

In order to address different elements in the topic map, Toma provides path expressions. The path expressions are chained using a dot. Toma path expressions might have an input, an output and a result value.

Input: The input is what comes to the left of the path expression.
For example, in the path expression `$topic.name.var`, `$topic` is the input of `.name` and `$topic.name` is the input of `.var`.

Output: The output is the result of the path expression itself. For example, in the path expression `$topic.name.var`, the output of `$topic.name` is the name branch and therefore it is possible to ask about its variant.

Result: The result value is the textual representation of the output. For example, the result value of a name, is the name value. The result value is taken into account only if this expression is the last (right-most) expression in the path. For example, the result value of the expression `$topic.name.var` is the actual variant of the name of the topic `$topic`.

Path expressions can work with sequences. For example, a topic might have more than one name. In that case, the path expression `$topic.name` is resolved to a sequence of names. Example:

```
select $topic.name where $topic='cpu';

 $topic.name
------------------------
central processing unit
central processor
processor
```

2.1 Elementary Path Expressions

Elementary path expressions are accessors that refer to a specific element in the topic map. The following table gives an overview.

Expression	Input	Output	Result
`.id`	topic / name / variant / occurrence / association	locator: Item identifier of the input	a string which is the value of the locator
`.si`	topic	locator: subject identifier of the input	a string which is the value of the locator
`.sl`	topic	locator: subject locator of the input	a string which is the value of the locator
`.name`	topic	name: name of the input	name value as a string
`.var`	name	variant of the input	the value of the variant as reference to an information resource (IRI) or as the resource itself – as a string.
`.oc`	topic	occurrence of the input	the value of the occurrence as reference to an information resource (IRI) or as the resource itself – as a string
`.ref`	occurrence / variant	locator: the reference to the information resource that is kept in the occurrence or variant	a string which is the value of the locator
`.data`	occurrence / variant	string: the data itself	the same as the output
`.sc`	name / variant / occurrence / association	a topic which is the scope of the input	a string which is the value of the locator which is the item identifier of the output topic.
`.player`	association	a topic which is a player of the input	a string which is the value of the locator which is the item identifier of the output topic.
`.role`	association	a topic which is a role of the input	a string which is the value of the locator which is the item identifier of the output topic.
`.reifier`	name / variant / occurrence / association	a topic which reifies the input	a string which is the value of the locator which is the item identifier of the output topic.

The sub-sections below detail and provide examples for each accessor.

2.1.1 item identifier (`.id`)

Input	topic / name / variant / occurrence / association
Output	locator: Item identifier of the input
Result	A string which is the value of the locator

Examples:

```
$topic.id
$topic.name.id
$topic.name.var.id
$topic.oc.id
$association.id
```

2.1.2 subject identifier (`.si`)

Input	Topic
Output	locator: subject identifier of the input
Result	a string which is the value of the locator

Example:

```
$topic.si
```

2.1.3 subject locator (`.sl`)

Input	topic
Output	locator: subject locator of the input
Result	a string which is the value of the locator

Example:

```
$topic.sl
```

2.1.4 name (`.name`)

Input	topic
Output	name: name of the input
Result	name value as a string

Example:

```
$topic.name
```

Extending path expressions for `.name`:

```
$topic.name.var
$topic.name.sc
```

2.1.5 variant (`.var`)

Input	name
Output	variant of the input
Result	the value of the variant as reference to an information resource (IRI) or as the resource itself – as a string.

Example:

```
$topic.name.var
```

Extending path expressions for `.var`:

```
$topic.name.var.ref
$topic.name.var.data
```

2.1.6 occurrence (.oc)

Input	topic
Output	occurrence of the input
Result	the value of the occurrence as reference to an information resource (IRI) or as the resource itself – as a string

Example:

```
$topic.oc
```

Extending path expressions for .oc:

```
$topic.oc.ref
$topic.oc.data
```

2.1.7 reference (.ref)

Input	occurrence / variant
Output	locator: the reference to the information resource that is kept in the occurrence or variant
Result	A string which is the value of the locator

Example:

```
$person.name.var.ref
$person.oc.ref
```

2.1.8 data (.data)

Input	occurrence / variant
Output	string: the data itself
Result	the same as the output

Examples:

```
$person.name.var.data
$person.oc.data
```

2.1.9 scope (.sc)

Input	name / variant / occurrence / association
Output	a topic which is the scope of the input
Result	a string which is the value of the locator which is the item identifier of the output topic

Example:

```
$person.name.sc
$person.name.var.sc
$person.oc.sc
$association.sc
```

2.1.10 Player (`.player`)

Input	association
Output	a topic which is a player of the input
Result	a string which is the value of the locator which is the item identifier of the output topic

Example:

```
$association.player
```

2.1.11 Role (`.role`)

Input	association
Output	a topic which is a role of the input
Result	a string which is the value of the locator which is the item identifier of the output topic

Example:

```
$association.role
```

2.1.12 reifier (`.reify`)

Input	name, variant, occurrence, association
Output	a topic that reifies the input
Result	a string which is the value of the locator which is the item identifier of the output topic

Example:

```
$topic.name.reifier
$topic.name.var.reifier
$topic.oc.reifier
$association.reifier
```

2.2 Instantiation

There are two path expressions that refer to the instantiation hierarchy defined in the topic map. The following table gives an overview.

Expression	Input	Output	Result
`.type`	topic	The topic that is the type of the input	a string which is the value of the locator which is the item identifier of the output topic
`.instance`	topic	The topic that is the instance of the input	a string which is the value of the locator which is the item identifier of the output topic

Note that brackets can be used after all of these expressions to indicate the level of the type. See section 2.6.1 on page 21.

2.2.1 Type (`.type`)

Input	Topic
Output	the topic that is the type of the input
Result	a string which is the value of the locator which is the item identifier of the output topic

Example:

```
$topic.type
```

Note that brackets can be used after the .type expression to indicate the level of the type. See section 2.6.1 on page 21.

2.2.2 Instance (`.instance`)

Input	Topic
Output	the topic that is the instance of the input
Result	a string which is the value of the locator which is the item identifier of the output topic

Example:

```
$topic.instance
```

Note that brackets can be used after the .instance expression to indicate the level of the type. See section 2.6.1 on page 21.

2.3 Inheritance

There are two path expressions that refer to the inheritance hierarchy defined in the topic map. The following table gives an overview.

Expression	Input	Output	Result
`.super`	topic	the topic that is the super-class of the input	a string which is the value of the locator which is the item identifier of the output topic
`.sub`	topic	the topic that is the sub-class of the input	a string which is the value of the locator which is the item identifier of the output topic

Note that brackets can be used after all of these expressions to indicate the level of the type. See section 2.6.1 on page 21.

2.3.1 Super-class (`.super`)

Input	Topic
Output	the topic that is the super-class of the input
Result	a string which is the value of the locator which is the item identifier of the output topic

Example:

```
$topic.super
```

Note that brackets can be used after the .super expression to indicate the level of the type. See section 2.6.1 on page 21.

2.3.2 Sub-class (`.sub`)

Input	Topic
Output	the topic that is the sub-class of the input
Result	a string which is the value of the locator which is the item identifier of the output topic

Example:

```
$topic.sub
```

Note that brackets can be used after the .sub expression to indicate the level of the type. See section 2.6.1 on page 21.

2.4 Associations

2.4.1 The Association Expression

An association path expression is written as follows:

```
association_id(association_type)->(role)
```

The whole path expression is resolved to a topic playing the given role in an association of the given type. The association type and the role should all be expressions that are resolved to topics. If the association id is not needed, it can be omitted. However, in case it is omitted, two different expressions can refer to two different associations. For example:

```
    (host-location)->(host) = $h
and (host-location)->(location) = $l
```

In the two expressions above, `$h` and `$l` can be players of different associations of type host-location. If we want to refer to two players of the same association, we can use the same association variable in both expressions:

```
    $a(host-location)->(host) = $h
and $a(host-location)->(location) = $l
```

Note that a much better approach is to chain the two players, as is described later. Note also that in the following example, `$a` and `$b` can but do not have to refer to the same association. If those associations have to be different, then the condition `$a != $b` must be added:

```
    $a(host-location)->(host) = $h
and $b(host-location)->(location) = $l
and $a != $b
```

The main feature of the association path expression is that it has no input, thus it starts the path expression (much like a topic literal or a variable). Although this feature by itself is sometimes very useful, it can be a major disadvantage when trying to refer to a chain of associated topics as demonstrated in the following example. In this quite complex example, we refer to a rather simple chain of associations: all the topics that are connected to a topic which is connected to the topic “finger”:

```
    $a(connect_to)->($r1) = i'finger'
and $a(connect_to)->($r2) = $middle
and $b(connect_to)->($r3) = $middle
and $b(connect_to)->($r4) = $topic
and $a != $b
```

Another disadvantage is the awkward way in which one must control the expressions using the association variables in such a chain (here we have to state that `$a` should not be equal to `$b`). These disadvantages have been solved by introducing the left arrow as described in the next section.

Some example queries containing associations:

Example 1:

```
select $association
  where $association.id = 'part-whole';
```

This example does not deal with associations at all. Instead, it comes to emphasis that the name of a variable has no effect on the semantics of a statement: the engine does not know that the variable `$association` is an association. It deals with it as if it were an ordinary topic variable and then looks among all topics which one has the id 'part-whole'.

Example 2:

```
select $association
  where $association(part-whole)->(part) = 'cpu'
    and $association.sc = 'functional';
```

In this example, the engine knows that `$association` is an association variable, because `$association` is placed in the association path expression in the *association_id* position. This variable can then be used later in the query to narrow down the result. Here it will look among all associations with type 'part-whole' for an association that has the topic with id 'functional' as a scope and that has the topic with id 'cpu' as a member within the role 'part'.

Example 3:

```
select $topic
  where $a(superclass-subclass)->(superclass) = $topic
  and $b(superclass-subclass)->(subclass) = $topic
  and $a != $b;
```

In this example a topic is searched for that is member of two associations: the topic has to have a role superclass in the first association `$a` of type 'superclass-subclass' and has to have a role subclass in the second association `$b` of type 'superclass-subclass'. Also the two associations have to be different.

2.4.2 Chaining Players

When you want to describe a long chain of associations, you will find that it is difficult to do this with the ordinary path expression as described in the previous chapter, as you can see in the following example:

```
select $topic
 where $a(connect_to)->(connected) = 'little_finger'
 and $a(connect_to)->(connected) = $p1
 and $p1 not in ('little_finger')
 and $b(connect_to)->(connected) = $p2
 and $b(connect_to)->(connected) = $p1
 and $p2.name != $p1.name
 and $c(connect_to)->(connected) = $p
 and $c(connect_to)->(connected) = $p2
 and $p.name != $p2.name
 and $a != $b
 and $b != $c;
```

Here we want to find the topics that are connected to ‘little_finger’ **via two other topics**. It can be seen that the statement describing this relationship is quite complex.

Therefore the association path expression described before is extended to include also the role of the input player in the same association in order to chain players:

```
.(role1)<-association_id(association_type)->(role2)
```

This path expression is resolved to a player that plays the *role2* in the association of type association type where the input player (the one coming to the left of the association path expression) plays the role *role1* in the very same association. If more than one such association path expression is chained, the associations of two consecutive players are never the same.

Now, the example above can be rewritten to:

```
select $topic
  where id('little_finger').($$)<-(connect_to)->($$)
                           .($$)<-(connect_to)->($$)
                           .($$)<-(connect_to)->($$) = $topic;
```

Note the anonymous variables (described in section 1.8 on page 7) for the roles, as we are not interested in those.

2.5 Square Brackets

The output of a path expression can be either empty, one element or a sequence. In the following example, the number of names that are returned by `$topic.name` is determined by the number of names that are defined in the topic map for the topic mouse:

```
select $topic.name where $topic.id = 'mouse';
```

Square brackets that contain a quoted string containing the value of **a chosen item** are used to specify that item out of the result sequence and can come after any path expression. For example:

```
$topic.name['central processing unit'] # the name 'central
                                       # processing unit' of
                                       # the topic $topic.
```

Another example:

```
 select $topic,
        $topic.id['cpu'],
        $topic.name['Central Processing Unit'],
        $topic.name['Central Processing Unit'].var['CPU']
  where exists $topic;

$topic | $topic.id['cpu'] | $topic.name['Central | $topic.name[
       |                  | Processing Unit']    | 'Central
       |                  |                      | Processing
       |                  |                      | Unit'].var['CPU']
-------+------------------+----------------------+-----------------
 cpu   | cpu              | Central Processing   | CPU
       |                  | Unit                 |
(1 row)
```

Another way to specify the items within a sequence is to use **a variable within** the square brackets:

```
$topic.name[$name] # $name will get the values of the sequence.
                   # we can use $name in another place to limit
                   # the sequence.
```

This lets us control the sequences in a better way. For example we can select only variants of names of topic foo starting with 'a':

```
select $topic.name[$name].var
 where $topic.id = 'foo' and $name ~ '^a';
```

Square brackets also provide a way to access **intermediate players** in a long **chain of associations**. For example, the following statement returns all the possible paths between the topic 'stomach' and the topic 'insulin' through exactly three associations, wherein the first is of type connect to.

```
   select 'stomach', 'connect_to', $p1, $at1, $p2, $at2, 'insulin'
    where 'stomach' >> id.($$)<-(connect_to)->($$)[$p1]
          .($$)<-($at1)->($$)[$p2]
          .($$)<-($at2)->($$) = 'insulin';


'stomach'|'connect|$p1     | $at1  | $p2    | $at2  |'insulin'
         |_to'    |        |       |        |       |
---------+--------+--------+-------+--------+-------+---------
 stomach |connect_|duodenum|connect|pancreas|produce| insulin
         |to      |        |_to    |        |       |
(1 row)
```

Note the use of the anonymous variable for the roles (because we do not need the roles in any other place).

2.6 Round Brackets

Round brackets are used for indicating the level of instantiation or inheritance but they are also used as typing brackets, to indicate a type.

In addition, brackets can be used within a path expression in order to group expressions and to control precedence.

For example:

```
($association_class.instance(1))->($role_class.instance(1)).name
```

The above returns the names of a player in the association. The type of the associations is any instance of `$association_class`. The role of the player is any instance of `$role_class`.

2.6.1 Level[1]

One can add a level to the path expressions for inheritance and instantiation. An overview is given in the following table.

Expression	Input	Output	Result
`.type(LEVEL)`	Topic	The topic that is the type of the input at the chosen levels	Topic
`.instance(LEVEL)`	Topic	The topic that is the instance of the input at chosen levels	Topic
`.super(LEVEL)`	Topic	The topic that is the super-class of the input at the chosen level(s)	Topic
`.sub(LEVEL)`	Topic	The topic that is the sub-class of the input at the chosen level(s)	Topic

For more explanation about the different path expressions displayed above, see Instantiation (section 2.2 on page 14) and Inheritance (section 2.3 on page 15).

The LEVEL parameter can be any non-negative number, an asterisk or a range. Level zero means the topic itself, level one means the type / instance / super / sub of the topic, level two indicates the type / instance / super / sub of the type / instance / super / sub of the topic and so on. An asterisk is used to refer to any level including zero. A plus is used to refer to any level excluding zero. A range, for example `1..*`, is defined by a non-negative number, two dots and a greater number or an asterisk. Examples:

```
$topic.type(1)      # the direct type of $topic (equivalent to
                    # $topic.type)
$topic.type(2)      # the type of the type of $topic
$topic.type(*)      # the types at any level of $topic
                    # including $topic itself
$topic.type(2..*)   # the types at any level of $topic without the
                    # direct type and $topic itself
$topic.type(0)      # gives $topic itself
```

Similar examples apply to the other expressions.

Another example:

```
$topic.type(1).super(*) # any direct type of the topic, or any
                        # parent (through superclass-subclass
                        # association) of that direct type.
```

[1] Maybe the concept of levels in respect to the types and super types is not really needed. If we remove it the brackets are used only to mark types with path expressions (and not levels).

2.6.2 Type

Apart from using brackets to indicate the level of the inheritance and instantiation path expressions as explained above, brackets are also used to indicate a type. Brackets can follow several path expressions (see section 2 on page 8) or precede the association arrow in the association expression (see section 2.4.1 on page 16). In such cases they allow to specify the types of the elements. The following table gives an overview.

Name	`.name(name-type)`
occurrence	`.oc(occurrence-type)`

association	`(association-type)->(role-type)`
	or
	`association_id(association-type)->(role-type)`
	or
	`.(role-type)<-(association-type)->(role-type)`
	or
	`.(role-type)<-association_id(association-type)->(role-type)`

The type itself can be specified as an expression that is resolved to a topic, that is, a topic literal (including a naked identifier), a variable containing a topic or a path expression that is resolved to a topic.

Examples:

```
.name(abbreviation)        # the name has to be of type abbreviation
$a(part-whole)->(part)     # the association $a has to be of type
                           # 'part-whole' and the role type is 'part'
$topic.oc(description)     # the occurrence has to be of type
                           # 'description'
```

2.7 The @ scope

The `@` symbol is used for specifying the scope of a name, an occurrence or an association. It also allows specifying the scope of a variant (which are described as parameters in XTM 1.0). The scope sign should be followed by an expression that is resolved to a topic and is written as shown in the following table:

name	`.name@scope` `.name(type)@scope`
variant	`.var@scope`
occurrence	`.oc@scope` `.oc(type)@scope`
association	`association_id(association_type)@scope->(role)`

The @ sign always follows the brackets that indicate the type of the expression

For Example:

```
$topic.name@en          # the name of the topic in the
                        # English scope
$topic.oc(size)@metric  # the occurrence of type 'size'
                        # in the metric scope
```

2.8 Precedence

Toma path expressions are evaluated **from left to right**.

For example in the expression `$topic.oc(description)@en.rd`, `$topic` is evaluated first. Then its occurrence of type description in the scope 'en' is evaluated. Finally the resourceData of this occurrence is returned.

Expressions in **brackets** (including those in typing brackets – see section 2.6.2 on page 22) have higher precedence. For example, in the expression `$topic.oc($a.type)@en.rd`, the type of the occurrence is evaluated as a whole (the type of the variable `$a`) before the occurrence is evaluated.

3 Toma statements

This chapter presents the statements that can be used in Toma.

In the notation presenting the syntax of the different statements, any clause surrounded by square brackets is optional. In addition, curly brackets are used to group possible options and a vertical bar symbol ("|") is used as disjunction between those options.

3.1 The SELECT statement

The SELECT statement is used in order to define queries over topic maps. The SELECT statement syntax is as follows:

```
SELECT [ ALL | DISTINCT ] navigation_list
[ WHERE clause ]
[ { UNION | INTERSECT | EXCEPT } [ ALL ] other_select ]
[ ORDER BY expr1 [ ASC | DESC ] [, expr2 [ ASC | DESC ] ...] ]
[ LIMIT integer ]
[ OFFSET integer ];
```

In general, variables can be introduced in the WHERE clause but also in the SELECT clause. In the next chapters we describe each of these clauses of the SELECT statement.

3.1.1 The SELECT clause

The SELECT clause of a query is used to define projections over the values of the variables found in the WHERE clause. The SELECT clause defines how rows in the result look like, in a similar manner to the SELECT clause in SQL.

The navigation_list controls which values will be presented in the result set. It is possible to introduce new variables in the navigation list and to use any path expression.

The DISTINCT keyword will cause that no duplicated rows are returned.

The ALL keyword is the opposite of the DISTINCT keyword and is the default behaviour.

The result is arranged as rows in a similar manner to the results that are returned from relational databases. Any returned variable will be represented by the id of its value as shown in the following example.

```
select $topic, $topic.id where $topic.name.var = 'CPU';

$topic     | $topic.id
-----------+----------
processor  | processor
(1 row)
```

Note that the SELECT clause returns elements of the values that are already chosen in the WHERE clause.

For example:

```
select $topic.name where $topic.name = 'lung';

 $topic.name
-------------
 lung
 long
(2 rows)
```

In this example, we select all the topics that have the name lung. Then we ask to show the names of the topics we found. We find one topic that has the two names lung and long. So the `$topic.name` in the SELECT clause means that we want to see all the names of the topic objects that can be $topic according to the WHERE clause.

This behaviour is a useful feature. For example, if we need to “translate” names between scopes we can write:

```
select $topic.name@dutch where $topic.name@english = 'lung';

$topic.name@dutch
-----------------
long
(1 row)
```

The ability to include new variables in the selection part allows generating one column which is totally dependent on the value of another column. For example, it is possible to retrieve the scope of the name that is shown in the result set as follows:

```
select $topic.name@$scope, $scope.id where $topic.name = 'lung';

$topic.name | $scope.id
------------+----------
lung        | english
long        | dutch
(2 rows)
```

In this example, `$topic` is set in the WHERE clause, and then its names are listed in the SELECT clause. However, the `$scope` variable is introduced in the selection clause as any scope of the listed names of `$topic`. Therefore, $scope gets a value for each row in the result set.

3.1.2 The WHERE clause

The WHERE clause may contain the following sub-clauses:

- clause AND clause
- clause OR clause
- EXISTS expr
- expr1 Comparison expr2
- NOT clause
- expr1 IN (expr, ...)

These sub-clauses are explained in the next sub-sections:

3.1.2.1 The EXISTS sub-clause

The syntax for the EXISTS sub-clause is as follows:

```
EXISTS path_expression
```

where *path_expression* is a Toma path expression as described in section 2 on page 8.

If the result of evaluating the path expression is not empty, the sub-clause is evaluated to true.

Example:

```
select $topic
 where exists $topic.si;
```

3.1.2.2 The comparison sub-clauses

A comparison sub-clause consists of two expressions and a comparison operator between them. The syntax is described as follows:

```
expression1 = | != | ~ | ~* | !~ | !~* expression2
```

In this definition an expression might be any path expression, a variable or a quoted string.

When using the regular expression comparison operators, both expressions must be evaluated to a string.

A comparison operator can be one of the following:

=	Equality operator. The two expressions around the equal sign should be equal to each other.
!=	Inequality operator. Negation is used in Toma as filtering (unlike in SQL). Therefore, the inequality operator is used to filter out any equality between the two expressions.
~	Case sensitive regular expression match operator. The regular expressions that can be used are Perl-like regular expression. See Perl Compatible Regular Expressions (PCRE)[2] for details.
~*	Case insensitive regular expression match operator.
!~	Negation of ~.
!~*	Negation of ~*.
IS NULL	Has the same meaning as NOT EXISTS.
IS NOT NULL	Has the same meaning as EXISTS.

3.1.2.3 The NOT sub-clause

The NOT sub-clause consists of one search condition. Its syntax is described as follows:

```
NOT formula
```

In the notation above *formula* can be any of the WHERE sub-clauses.

In Toma negation is used for filtering. Thus the negation adds constraints on the value of the variables in the WHERE clause.

3.1.2.4 The AND sub-clause

The AND sub-clause consists of two search conditions. Its syntax is described as follows:

```
formula1 AND formula2
```

In order for the AND sub-clause to be evaluated to true, both formulas in the above notation should be evaluated to true - that is, the variables in those formulas should get values so that both formulas are true. If there are no such values, the AND sub-clause is evaluated to false.

3.1.2.5 The OR sub-clause

The OR sub-clause consists of two search conditions. Its syntax is described as follows:

```
formula1 OR formula2
```

In order for the OR sub-clause to be evaluated to true, at least one of the formulas in the above notation should be evaluated to be true - that is the variables in the formulas will get all the possible values so that at least one of the formulas is true. If there are no such values, the OR sub-clause is evaluated to false.

3.1.2.6 The IN sub-clause

The IN sub-clause consists of an expression, the IN keyword and a list of comma separated expressions within brackets. Its syntax is described as follows:

```
expression IN ( expression1, expression2 ... )
```

The IN sub-clause is evaluated as:

```
(expression = expression1 OR expression = expression2 OR ... )
```

Instead of the list of expressions you can also insert a sub select statement:

```
expression IN (sub-select-statement)
```

The selection part of the sub select should be of exactly one expression. Also, it is forbidden to use variables from the main SELECT statement in the sub select statement.

Example:

```
# all the names of the topics of type
# 'mechanical device' which have an occurrence of type
# 'mass' which is equivalent to one of the occurrences
# of the same type of topics of type 'pc card'
select $topic1.name
  where $topic1.type.name = 'mechanical device'
  and $topic1.oc(mass) in (select $topic2.oc(mass)
                           where $topic2.type.name = 'pc card');
```

3.1.3 The UNION clause

The syntax of the UNION clause is as follows:

```
select1 UNION [ ALL ] select2
```

UNION appends the result set of the first SELECT statement (*select1*) to the result set of the second SELECT statement (*select2*). It also eliminates all duplicates (so it runs DISTINCT on the result) unless the ALL keyword is used.

In order for UNION to work, both selects must have similar selection clauses: they have to have the same number of expressions, and each expression in one select has to resolve to the same type (topic, association, name etc.) of the expression in the other select in the same position.

3.1.4 The INTERSECT clause

The syntax of the INTERSECT sub-clause is as follows:

```
select1 INTERSECT select2
```

INTERSECT returns all the results that are in both result sets.

Both selection clauses must be similar: they have to have the same number of expressions, and each expression in one select has to resolve to the same type (topic, association, name etc.) of the expression in the other select in the same position.

3.1.5 The EXCEPT clause

The syntax of the EXCEPT sub-clause is as follows:

```
select1 EXCEPT select2
```

EXCEPT returns all the results that are in the first SELECT statement (*select1*) but not in the second SELECT statement (*select2*).

Both selection clauses must be similar: they have to have the same number of expressions, and each expression in one select has to resolve to the same type (topic, association, name etc.) of the expression in the other select in the same position.

3.1.6 The ORDER BY clause

The ORDER BY clause controls the way the result is ordered. The syntax of the ORDER BY sub-clause is as follows:

```
ORDER BY column_number [ ASC | DESC | NASC | NDESC ]
         [, column_number [ ASC | DESC | NASC | NDESC ] ...]
```

The list of the column numbers which follows the ORDER BY keywords defines ordering constraints over the variables used in the SELECT clause. The first column can be referred to as number one, the second, as number two etc.

Each *column_number* can be succeeded by one of the keywords ASC (ascending), DESC (descending), NASC (numerical ascending) or NDESC (numerical descending). ASC is the default. ASC and DESC are for ordering alphabetically (so 10 comes before 2). NASC and NDESC are for ordering numerically. In that case, any value that is not a number is resolved to 0.

3.1.7 The LIMIT clause

The LIMIT clause provides a way to retrieve only a portion of the result. The syntax of the LIMIT sub-clause is as follows:

```
LIMIT integer
```

In the above notation integer specifies the total number of rows to be retrieved.

3.1.7.1 The OFFSET clause

The OFFSET clause provides a way to retrieve only a portion of the result. The syntax of the OFFSET sub-clause is as follows:

```
OFFSET integer
```

In the above notation integer controls the row to start from.

4 Toma Functions

4.1 String Functions

The operator and functions in this section can be used only on strings. Numbers are converted to strings using default conversion[2].

4.1.1 The double pipe operator

The double pipe operator || concatenates two strings.

Example:

```
select $topic
  where $topic.name = 'cp' || 'u';
```

4.1.2 LOWERCASE(string)

This function converts all characters in the string to lower case characters.

Example:

```
select $topic.oc(description), lowercase($topic.oc(description))
  where $topic.id = 'cpu';

 $topic.oc(description) | lowercase($topic.oc(description))
------------------------+-----------------------------------
 The CPU is the brains  | the cpu is the brains
 of the computer.       | of the computer.
(1 rows)
```

4.1.3 UPPERCASE(string)

This function converts all characters in the string to upper case characters.

Example:

```
select $topic.oc(description), uppercase($topic.oc(description))
  where $topic.id = 'cpu';

 $topic.oc(description) | uppercase($topic.oc(description))
------------------------+-----------------------------------
 The CPU is the brains  | THE CPU IS THE BRAINS
 of the computer.       | OF THE COMPUTER.
(1 rows)
```

[2] The details of this conversion has still to be determined.

4.1.4 TITLECASE(string)

This function converts all characters to lower case except for the initial characters which are converted to upper case characters.

Example:

```
select $topic.oc(description), titlecase($topic.oc(description))
  where $topic.id = 'cpu';

 $topic.oc(description) | titlecase($topic.oc(description))
------------------------+----------------------------------
 The CPU is the brains  | The Cpu Is The Brains
 of the computer.       | Of The Computer.
(1 rows)
```

4.1.5 LENGTH(string)

This function returns the length of a string.

Example:

```
select $topic.oc(description), length($topic.oc(description))
  where $topic.id = 'cpu';

 $topic.oc(description) | length($topic.oc(description))
------------------------+----------------------------------
 The CPU is the brains  | 38
 of the computer.       |
(1 rows)
```

4.1.6 SUBSTR(string, from, [length])

Provides the ability to retrieve a specific part of the string. It returns a sub-string of *string* starting from the *from* character (the first character is at index 1). If *length* is provided, the returned string will be of that length.

Example:

```
select $topic.oc(description), substr($topic.oc(description),7,11)
  where $topic.id = 'cpu';

 $topic.oc(description) | substr($topic.oc(description),7,11)
------------------------+-----------------------------------
 The CPU is the brains  | U is the br
 of the computer.       |
(1 rows)
```

4.1.7 TRIM(string, [LEADING | TRAILING | BOTH], [characters])

This function trims the string. It removes occurrences of any character from the start and/or end of string. If *LEADING*, *TRAILING* or *BOTH* are not provided, *BOTH* is taken as the default. If *characters* is not provided, space is taken as the default. If it is provided, space is automatically included and it is case sensitive and can also contain symbols (like a dot).

Example 1:

```
select $t.oc(description), trim($t.oc(description), BOTH, 'hrTc.e')
  where $t.id = 'cpu';

 $t.oc(description)     | trim($t.oc(description), BOTH, 'hrTc.e')
------------------------+-----------------------------------------
 The CPU is the brains  | PU is the brains
 of the computer.       | of the comput
(1 rows)
```

Note that the lowercase ‘t’ in ‘computer’ is not trimmed because it is not in uppercase and so it does not match capital ‘T’.

Example 2:

```
select $topic.oc(description),
       trim($topic.oc(description),leading,'hrTc.e')
  where $topic.id = 'cpu';

 $topic.oc(description) | trim($topic.oc(description), leading, 'hrTc.e')
------------------------+-------------------------------------------------
 The CPU is the brains  | PU is the brains
 of the computer.       | of the computer.
(1 rows)
```

Example 3:

```
select $topic.oc(description),
       trim($topic.oc(description), trailing, 'hrTc.e')
  where $topic.id = 'cpu';

 $topic.oc(description) | trim($topic.oc(description), trailing, 'hrTc.e')
------------------------+--------------------------------------------------
 The CPU is the brains  | The CPU is the brains
 of the computer.       | of the comput
(1 rows)
```

4.2 Conversions Functions

4.2.1 TO_NUM(text)

This function converts text to a number if possible. If not possible it will be converted to 0.

Example:

```
select $topic.oc(mass), to_num($topic.oc(mass))
  where $topic.id = 'computer';

 $topic.oc(mass) | to_num($topic.oc(mass))
-----------------+----------------------------------------
 3.4 kg          | 3.4
(1 rows)
```

4.3 Aggregation Functions

Aggregation functions can be used only in the selection clause of a SELECT statement. If an aggregation function is present in a selection clause, all the expressions of that selection clause must be aggregation functions (this is due to the fact that currently there is no grouping in Toma; if grouping turns out to be needed, it will be added to Toma in the future).

4.3.1 COUNT(expression)

This function counts the number of values in the result denoted by the expression.

Example:

```
select count($topic.oc(description))
  where $topic = 'cpu';
```

4.3.2 SUM(expression)

This function sums the TO_NUM conversions of the result set represented by the expression. If one of the values is converted to NULL, it is evaluated to zero by the SUM function.

Example:

```
select sum($topic.oc(mass))
  where $topic.type = 'device';
```

In this example the result will be the sum of all the masses of the occurrences of topics with type device.

4.3.3 MAX(expression)

This function returns the maximum value among the TO_NUM conversions of the result set represented by the expression. NULL is evaluated to zero by the MAX function.

Example:

```
select max($topic.oc(mass))
  where $topic.type = 'device';
```

Here the result will be the maximum of all the masses of the occurrences of topics with type device.

4.3.4 MIN(expression)

This function returns the minimum value among the TO_NUM conversions of the result set represented by the expression. NULL is evaluated to zero by the MIN function.

Example:

```
select max($topic.oc(mass))
  where $topic.type = 'device';
```

Here the result will be the minimum of all the masses of the occurrences of topics with type device.

4.3.5 AVG (expression)

This function calculates the simple average of values among the TO_NUM conversions of the result set represented by the expression. NULL is evaluated to zero by the AVG function.

Example:

```
select avg($topic.oc(mass))
  where $topic.type = 'device';
```

Here the result will be the average of all the masses of the occurrences of topics with type device.

4.3.6 CONCAT (expression, [string])

This function concatenates the values of the result specified by the expression. If *string* is defined, it is placed as a separator between the values.

Example:

```
select concat($topic, ', ')
  where $topic.type = 'device';
```

Here the result will be a comma-separated list of all the topics with type device.

5 Further Ideas and Missing Features

This chapter includes some features that are still missing in the above description, together with some draft suggestions of how to include these features. Note that all syntax in the examples below is very early draft.

5.1 Conversions Functions

5.1.1 TO_UNIT(text, target_unit)[3]

This function converts between units. The function assumes that the text contains a number and a unit indicator. This function is implemented by using the *Units Conversion Library* by *Mayo Foundation*. List of all possible units can be found in the site of the *Units Conversion Library*: http://sourceforge.net/docman/display_doc.php?docid=2810&group_id=19449#SECTd0e169.

Example:

```
select $topic.oc(mass), to_unit($topic.oc(mass), 'pound')
  where $topic.id = 'computer';

 $topic.oc(mass) | to_unit($topic.oc(mass), 'pound')
-----------------+-------------------------------------
 3.4 kg          | 7.49571641852906
(1 rows)
```

5.2 Conversions of different data types

There should be conversion functions between all the possible data types (such as defined in http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#built-in-primitive-datatypes). However, the initial intention is that the language provides default behaviour for dealing with different data types in a similar way that TO_NUM function is called over data that is not number when functions such as AVG or SUM are called.

5.2.1 Conversion between units

One attractive possibility is to set a default unit system for all the calculations, and let the engine make the conversions automatically. Something like:

```
use units = metric;
```

5.3 Arithmetics

All the basic arithmetics operations such as addition, subtraction, division and multiplication should be added to the functions chapter.

[3] But see also the suggestion in section 5.2.1 on page 36.

5.4 Prefixes

In order to be able to refer to a topic map, a prefix that describes its base should be set. Something like:

```
set other = http://site.somewhere.com/other/topic/map;
```

And later to use it as:

```
select $topic.oc(mass)
  where $topic.type = 'other:device';
```

The default prefix is *this*. It is set by itself, but can be set to other value then the default:

```
set this = http://site.somewhere.com/this/topic/map;
```

5.5 Scoping

It may be useful to set scoping within a session. The inclusion of arithmetics into the scoping can help when it is needed to detail exactly what is in and what is out with relation to the scoping:

```
scope by scope1 and scope2 and scope3;
```

All the statements following this statement will see a topic map where only constructs with scope1, scope2 and scope3 are available.

```
scope by scope1 or scope2 or scope3;
```

All the statements following this statement will see a topic map where only constructs with scope1, scope2 or scope3 are available.

In order to reset the scoping, the following statement will be used:

```
scope reset;
```

Another alternative is to include the above scoping logical arithmetics within the path expression.

```
$topic.name@(en and nl)
```

But probably this following does not make too much sense (especially in the SELECT clause):

```
$topic.name@($scope1 and $scope2)
```

This means that we should probably limit this kind of arithmetics to constants (and not variables).

5.5.1 Defining scoping by time

It is very useful to be able to assign validity periods for associations, occurrences and names. Currently the way to do that is to define scopes which scope these constructs.

The way this is done is by define topics that represent time intervals and use these topics to scope the constructs.

It will be useful if the language can support this kind of model. There are different approaches possible:

- No support for time arithmetics is provided by the language. Set of queries will identify the time interval topics that are relevant for certain time and these topics will be used for scoping the next queries.
- Support is provided by the language to define the way the time dependent data is modelled, as well as filtering the data by time.
- A standard is created for modelling time dependent data using scopes is created, and the language supports this standard.
- The TMDM standard is extended to support apart from scopes, also validity dependent on time.

Below, we describe how the second approach could be followed, although because time sequences become more and more important, we believe that the third or the fourth approaches are more appropriate.

If we assume that we need a scoping topic that has two occurrences defining start and end time stamps, we can define the following:

```
set time-scope-type = event;
set time-start-occurrence-type = start;
set time-end-occurrence-type = end;
```

That is, all the topics of type *event* will have two occurrences of type *start* and *end*, and these topics will define the time dependent validity of other constructs.

However, many times the time interval must include the information of weather it is positive or negative interval (i.e. weather the mean value relates to the past or to the future). In that case the following might be defined:

```
set time-scope-type = event;
set time-stamp-occurrence-type = timestamp;
set time-interval-occurrence-type = interval;
```

That is, all the topics of type *event* will have two occurrences of type *timestamp* and *interval*, and these topics will define the time dependent validity of other constructs. The occurrence of type *interval* will be able to have positive or negative values.

Now it is possible to write something like:

```
scope by time = '2005-10-30 T 10:45 UTC';
```

Note that this solution does not extend to topic maps that are modelled differently. The definition above can be extended to refer to specific topic maps, or we can resort to the third (or even better, the fourth) approach above.