ISO/IEC JTC 1/SC 34

Date: 2005-02-18

ISO/IEC WD 18048

ISO/IEC JTC 1/SC 34/WG 3

Secretariat: SCC

Topic Maps Query Language

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: International Standard
Document subtype:
Document stage: (20) Preparatory
Document language: E

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

ISO/IEC 18048 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information Technology*, Subcommittee SC 34, Document Description and Processing Languages.

Introduction

This International Standard defines a query language for Topic Maps known as TMQL (Topic Maps Query Language). This draft was informed by [TMQLuc][2] and [TMQLreq][1] and is for review for interested parties.

Ed. Note.
This is the first working draft, and it is quite rough. There are many internal inconsistencies (some intentional, some not), and the machinery used to specify the semantics of the language is not yet in place. For this reason the semantics are mostly expressed in prose. In later drafts the normative definition will be formalized, once the formal machinery has been finished.

Topic Maps Query Language

1 Scope

This International Standard defines a formal language by providing a syntax to form query expressions. The document also defines an informal and a formal semantics for every syntactic form, including rules for the reporting of error conditions.

To constrain the interaction with query processors, this International Standard also describes an abstract processing environment. In this part the passing of parameters into the query process and the exchange of result values is loosely defined. This environment also includes predefined functions and predicates every conformant processor must provide.

This International Standard does not define an API (application programming interface) nor does it define an extension mechanism for extending the predefined environment can be while, but it does provide means for importing external ontologies and functionality. This International Standard also remains silent on potential optimization techniques, leaving these for implementors and researchers to define.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

NOTE Each of the following documents has a unique identifier that is used to cite the document in the text. The unique identifier consists of the part of the reference up to the first comma.

Unicode, *The Unicode Standard, Version 3.0*, The Unicode Consortium, Reading, Massachusetts, USA, Addison-Wesley Developer's Press, 2000, ISBN 0-201-61633-5

TMDM, *ISO 13250-2 Topic Maps — Data Model*, ISO, 2005, available at <http://www.isotopicmaps.org/sam/sam-model/>

XML 1.0, *Extensible Markup Language (XML) 1.0*, W3C, Third Edition, W3C Recommendation, 04 February 2004, available at <http://www.w3.org/TR/REC-xml/>

RFC2396, *RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax*, The Internet Society, 1998, available at <http://www.ietf.org/rfc/rfc2396.txt>

3 Notation and Conventions

3.1 Notation and Syntax

The syntax of TMQL is defined using the EBNF formalism defined in [XML 1.0].

Additional convenience notations to extend the (minimal) grammar core are introduced via term equivalences. Variable parts in these terms are symbolized by variables which are implicitely all-quantified. These equations may be supported by additional prose.

For productions yet to be done the following pseudo nonterminal is used.

[0] *TBW* ::= 'TO_BE_WRITTEN'

3.2 Informal and Formal Semantics

The semantics of TMQL are fully defined in prose. As this may result in ambiguities, the prose is supported by several layers of formal and semi-formal mappings. SELECT expressions are completely mapped into FLWR expressions (Annex A). In Annex C TMQL FLWR expressions are mapped to TMQL path expressions (excluding the content constructor in RETURN clauses). TMQL path expressions are mapped to tau path expressions (Annex D). The latter is then mathematically formalized in Annex E. Annex F describes the relationship between the Tau model and TMDM.

4 Query Environment

Every implementation has to provide a query environment which allows applications to interact with the query processor.

TMQL query expressions are executed in an environment that contains information used by the query processor. When executing begins the environment may be empty, or it may contain initial values set by means outside the scope of this International Standard. The query environment provides the following input:

Base URI (optional)
An optional URI against which relative URIs in the query are resolved, if provided.

Variable bindings
A set of *variable bindings*, each of these being a name/value pair. A mechanism must exist to associate values to variables (or parameters) inside the compiled query. One such can be the *default topic map* which—if it is defined—must be a topic map item as defined in [TMDM]. The set is empty by default.

Prefix bindings
A set of (prefix, type, uri) tuples. The prefix is a string matching the 'ident' production, and it is an error if there are duplicate prefixes. The type is one of the strings 'indicator', 'source', 'subject', or 'module'. The uri is an absolute URI. The set is empty by default.

Ed. Note.
drrho: Not exactly sure, what it does, but I assume this includes all not-predefined functions, predicates, ontologies.....?.

larsbot: No, it's the prefixes used in foo:bar qnames, and only that.

Predicate bindings
A set of (name, predicate) tuples. The name is a string matching the 'ident' production, and it is an error if there are duplicate names. The set initially contains the predicates defined in 10.2.

Ed. Note.
rho: If we can merge 'using' and 'rules', so this can be merged too.

larsbot: They serve different syntactical functions, so that seems difficult.

Ed. Note.
rho: I also have concerns in hardcoding _tolog_ rules. What if TMCL can be used here? Or maybe we find a way to include OWL ontologies?

larsbot: Now that it's just "predicates" that should be possible, since in the environment one can find ways to turn TMCL/OWL constructs into TMQL predicates and get them into the initial environment. How does that sound?

This context must be provided by the query infrastructure and the application using it. How the context is generated or managed is outside the scope of this International Standard.

5 Types and Values

5.1 Content

The language supports various types of content; simple content such as numbers and strings or XML content, but also composite content like sequences of (tuples of) values, or Topic Map content.

[1] *content* ::= *simple-content* | *composite-content*

[2] *simple-content* ::= 'NULL' | *boolean* | *number* | *string* | *xml-content*

[3] *composite-content* ::= *tuple-content* | *tm-content*

The constant NULL represents the *undefined value*. It is typeless per se and is used for situations when no particular value should or can be used.

As an alternative we also allow undef to be used instead of NULL.

[A] 'undef' :== 'NULL'

Ed. Note.
larsbot: I think we should lose this. One syntax for null is enough, no matter which one it is.

5.2 Booleans

Boolean content consists only of the two constants FALSE and TRUE:

[4] *boolean* ::= 'FALSE' | 'TRUE'

FALSE is equivalent to NULL, but is different from TRUE.

Ed. Note.
larsbot: Do we really need this?

5.3 Numbers

The language supports constants denoting decimal numbers:

[5] *number* ::= '-'? [0-9_]+ ('.' [0-9_]+)?

The character _ (underscore) can be used to structure longer numbers and is otherwise ignored.

Ed. Note.
larsbot: I think we should lose this.

The language does not discern between decimal and integer numbers.

EXAMPLE Valid number constants are 3.14, 1_000_000 and -273.15, examples for invalid numbers are 3,14 (comma instead of dot) and - 273.15 (no blanks between the sign and the value are allowed).

Together with numbers, the unary operator - and the binary operators +, -, * and / are defined as usual, also with the usual precedence. The language also adopts the usual meaning for comparing numbers using =, !=, <, <=, > and >=.

Ed. Note.
drrho: Idea: use XSD and leave this discussion out of the standard?

larsbot: What would XSD allow us to leave out?

Ed. Note.
drrho: What operators to include?

larsbot: The listed set seems fine to me.

5.4 Strings

Strings are character sequences terminated by ":

[6] *string* ::= "" [^"]* ""

Where the following constraints hold:

1. the string value does not contain the terminating quotes,
2. any occurrence of the character " has to be escaped as \", and
3. the string value has every substring matching \u[0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f]([0-9A-Fa-f][0-9A-Fa-f])? replaced with the Unicode character whose code point is given in the substring in hexadecimal.

If a string contains substrings terminated with { and }, then these substrings are interpreted as TMQL value expressions (6.5). When strings are evaluated, first all these value expressions are evaluated. The results are then *stringified* according to the following rules:

1. Number content is converted into is textual representation.
2. String content is used as is.
3. XML content is serialized to a string representation.
4. Topic Map content is serialized into XTM.
5. Tuple sequence content is the concatenation of all tuple content. Tuple content is the concatenation of all serializations of tuple components.

If strings should contain the characters { and } verbatim, they have to be escaped as \{ and \}. For a string to contain the character " it must be escaped as \", and \ itself must be escaped as \\.

Together with strings, a concatenation operator + is defined. Also the comparison operators <, <=, > and >= with the usual meanings are defined.

EXAMPLE Valid strings are "abc" and "abc { 42 }def". Invalid strings are "That"s strange" (unescaped ") and "Something { RETURN 42 " (subquery not terminated).

5.5 Tuples

Tuples are ordered collections of heterogeneous components. The objects inside a tuple are the *components* and may only be of a simple type. The types of the tuple components can be organized also into a tuple, the *tuple profile*.

A tuple without a single component is called an *empty tuple*; it is equivalent with the constant NULL. Tuples with only a single component are called *singletons*. Any value of simple content can be interpreted as singleton.

The length of the tuple is called the *arity of the tuple*. The individual components of a tuple are ordered, the first component has assigned the index 0, the next 1, etc. The indices can be used to extract a component from a given tuple. If an index larger or equal the arity of a tuple is used the extraction will result in the value NULL.

When tuples are to be compared with each others, this is always done in the context of a *ordering tuple*. That tuple can only have components with values ASCENDING or DESCENDING. If that ordering tuple is missing, a tuple

containing only ASCENDING is assumed.

Two tuples can then be compared with each other using the following rules:

1. The empty tuple is smaller than any other tuple.

2. The comparison of non-empty tuples is done component-wise by starting with index 0 moving by one to the next higher index at a draw.

   If the order tuple has ASCENDING as value on a given index, that tuple with the smaller component on that index is also the smaller tuple. If the order tuple has DESCENDING as value on a given index, that tuple with the bigger component on that index is the smaller tuple.

3. The components at a given index are compared. If the components at this index are equivalent, then the components with the next higher index are investigated. In the latter case the tuple with the smaller arity is also the smaller tuple.

It is not possible to denote individual tuples.

5.6 Tuple Sequence

Tuple sequences are sequences of tuples where all tuples have identical profiles. They can be denoted as follows:

[7] *tuple-content* ::= '(' *tuple-sequence* ')'

[8] *tuple-sequence* ::= *tuple-sequence* '||' *tuple-sequence* |
value-expression order-direction ? (',' *value-expression order-direction* ?)+ |
'{' *value-expression* '}'

[9] *order-direction* ::= ('ASCENDING' | 'DESCENDING')?

Tuple sequences can be combined using the infix operator ||. For this, the tuple profiles of both tuple sequences must be identical, otherwise the compilation will fail with an type violation error.

The operator || will combine the two operands by interleaving any tuples into one result tuple sequence. Regardless of any ordering of the two operands, the resulting tuple sequence will be unordered.

The empty tuple sequence is equivalent with NULL. When compared, it is smaller than any other tuple sequence.

Ed. Note.
rho->rho: This is a bit flaky (again).

Ed. Note.
rho->rho: Embedding of other content into tuples is missing.

When a tuple is evaluated all components of the tuple are evaluated in no predefined order. The evaluation result of components denoting simple content will be interpreted as a sequence of length 1 with a singleton tuple containing this content. The evaluation of components denoted by path expressions may result in general tuple sequences. Only tuple sequences of singletons are valid, the evaluation will terminate with an evaluation exception if any of these sequences contain tuples which are not singletons.

Ed. Note.
If we want to confuse everyone, then we could allow tuples of tuples. Maybe leave it simple for now, although it would be mathematically cleaner.

The intermediary result of this tuple evaluation is then a tuple of sequences of singletons. This will be *flattened out* by building the cartesian product of all singleton sequences. The result sequence will only contain tuples with simple content.

Ed. Note.
This is soooo akward to describe in prose.

EXAMPLE 1 The tuple sequence (1, "two", %m//person) will return a tuple sequence where all tuples have as the first component the value 1 and on the second position the string "two". There will be as many tuples as there are topics being instances of person in the map %m.

Sequences can be ordered or unordered. If no order direction was provided in any of the tuple components, the result tuple sequence will be unordered. Otherwise the ordering information in the tuple will be used to construct the *order tuple* as follows:

1. If the order direction on a particular index is defined, this value will be used for the same index in the order tuple.

2. If no order direction was specified on a particular index, the default ASCENDING will be used as value for the same index in the order tuple.

That order tuple will be used for comparison according to 5.5.

EXAMPLE 2 The tuple sequence specified by (%m//person # bn DESCENDING) contains tuples with only a single component. That component contains all basenames of all instances of person in the map provided by %m. All these basename are sorted according to their (stringified) value in falling order.

5.7 XML Content

XML content follows exactly the syntactic rules given in the XML specification [XML 1.0] with one notable exception: XML content can contain value expressions (6.5) to generate more dynamic output. These subexpressions must be properly nested using a balanced pair of brackets { and } and can only occur in specific places:

— within an element tag,

— within an attribute value or as part of an attribute name,

— as part of element text.

Equivalently, if all occurrences of subexpressions (including the {} bracket pair) are replaced with the empty string, then the XML content must be well-formed.

[10] *xml-content* ::= *xml-element*

[11] *xml-element* ::= '<' *string* *xml-attribute* * '>' (*string* | *xml-element*)* '</' *string* '>' |
'<' *string* *xml-attribute* * '/>'

[12] *xml-attribute* ::= *string* '=' '"' *string* '"'

If the characters { and } are used as-is, they have to be escaped as \{ and \}.

EXAMPLE 1 The following XML content samples are valid:

— <copyright>Copyright Holder</copyright>

— <message>Celine Dion has quite a different sound than {$bn}.</message>

— <mess{$x}>This may work.</mess{$x}>

— <code lang="pascal">procedure TEST () \{ writeln; \}.</code>

The following XML content samples are invalid:

— <copyright>Copyright Holder (no end tag)

— <mess{$x}>This is broken.</{$y}age> (it depends on the values of $x and $y whether this is well-formed)

— <code lang="pascal">procedure TEST () { writeln; \}.</code> (opening { indicates subexpression)

When XML content is evaluated, first all subexpressions are evaluated in no predefined order. The content generated by these subexpressions is then embedded into the XML content, replacing the text of the subquery (including the {} bracket pair) according to the following embedding rules:

— Number content is converted into its string representation.

— String content is used as-is whereby special characters &, ", <, >, ' are automatically encoded to the predefined entities &, ", <, >, '.

— XML content is used as-is.

— Tuple content is stringified by concatenating the textual representation of all tuple components.

— TM content is serialized using XTM.

EXAMPLE 2 Given the following code,

```
LET $s := "Portishead"
    LET $x := <some>XML code</some>
    LET $m := <message>{$s} has no idea about {$x}.</message>
```

after evaluation $m will contain the XML content <message>Portishead has no idea about <some>XML code</some>.</message>.

5.8 Topic Map Content

Ed. Note.
rho->rho: This broke by shifting things around. Fix later.

The language supports to address whole topic maps or fragments thereof. It is also possible to directly denote topic map content:

[13] *tm-content* ::= *tm-content* '+' *tm-content* |
referenced-tm-content |
denoted-tm-content |
path-expression

Topic Map content can be either referenced or directly denoted. The operator + symbolizes the merge operations as defined in [TMDM]. URIs are defined as strings following the syntax in [RFC2396]. Finally, Topic Map content can also be computed using path expressions.

References can be interpreted as *subject identifiers*, *subject indicators* or *source locators*. For this purpose, it is optionally possible to qualify references with i, s or l:

[14] *referenced-tm-content* ::= (('i' | 's' | 'l') '"')? *tm-reference*

[15] *tm-reference* ::= *uri* | (*prefix* ':')? *identifier*

[16] *uri* ::= *string*

[17] *prefix* ::= *identifier*

If a reference is not qualified, it is assumed that the reference is a subject identifier. References can be either absolute (as URIs) or relative using an optional prefix and an identifier.

The prefix is supposed to be associated with a namespace (see 6.2.1), the URI of which will be used to resolve the identifier to an absolute URI. If a prefix is provided but not associated, a URI resolution error is flagged. During evaluation all URIs are resolved against the URI of the current context map. If this resolution does not result in an absolute URI, then an URI resolution error is flagged. The resulting URI will be resolved at evaluation time. The result of this evaluation is a Topic Map item, whereby implementations are allowed to postpone the actual loading of the data.

EXAMPLE 1 Given the URLs http://server/opera/ and http://server2/librettos/ refer both to topic maps, then topic map objects can

be referred to via

```
LET  := http://server/opera/
    LET    := http://server/opera/ + http://server2/librettos/
```

Apart from complete maps, also Topic Map items can be referred to, be they relative to the current context map or be they from some other, external map. Assuming that such topics can also reify functions and predicates, this mechanism can also be used to refer to these.

EXAMPLE 2 The path expression http://example.org/map//* addresses all information items in the specified map.

If other references are used, those references which are already absolute URIs will be used as is. References with a prefix identifier are translated by replacing the prefix (together with the colon :) with the value of the prefix. If the resulting URI is relative, it will be resolved against the URI of the context map. This also applies to references which consist of an identifier only. The resulting URI is use either as subject identifier, subject indicator or source locator to address a topic map item.

To address all topic and association items of a particular class inside a map we allow the following notation:

[B] '//' *identifier* :== '//*' '[' '*' *identifier* ']'

Ed. Note.
rho->rho: does not fit here

Ed. Note.
Lars: Define! Also add support for URI-based references.

rho: should be done now, plz check

rho: this is not perfect....hmm

Ed. Note.
Lars: Need to describe how this is mapped to a value. Involves namespaces. Want to do it using a core expression with a literal.

rho: maybe this should be more elaborated.

TM content can also be constructed:

[18] *denoted-tm-content* ::= *TBW*

Ed. Note.
Here we will need a good notation for creating topic maps.

TBW, tm-items, (topic/assoc), topic-identifiers, TMDM, topicidentifier, global (subject identifier), local tid, via subject identifier, where should this go???, //* gives tuple sequence of tm items.

6 Query Expressions

6.1 Structure

A query can take one of three forms: a SELECT expression, a FLWR expression, or a path expression:

[19] *query-expression* ::= *declaration* *
(*path-expression* | *select-expression* | *flwr-expression*)
order-clause ?
unique-clause ?

[20] *declaration* ::= *using* |
import |
ontology-declaration |

function-declaration

Any number of declarations can be put at the front of a query. These declarations apply through the whole query. Declarations are used to define functions, predicates, and name prefixes to be used in the query, and also to import modules containing functions and predicates.

SELECT and FLWR expressions both make use of path expressions as a sub-language. It is also possible to nest queries and mix the different styles.

6.2 Declarations

6.2.1 Prefix Declarations

The using declaration creates a URI prefix that can be used to create short and simple references to topic map objects using qualified URIs. It has the following syntax:

[21] *using* ::= 'using' *prefix* 'for' *referenced-tm-content*

Each using production causes a new tuple (prefix, type, uri) to be added to the prefix bindings in the query environment, where prefix is the string matching the 'ident' production; type being the i, s, or l in the referenced-tm-content; and uri the string matching the 'uri' production. It is an error if the prefix already exists in the prefix bindings.

EXAMPLE 1 The declaration below makes it possible to refer to topics with subject identifiers defined in XTM 1.0 using the xtm prefix.

```
using xtm for i"http://www.topicmaps.org/xtm/1.0/core.xtm#"
```

The reference xtm:sort would now refer to the XTM 1.0 sort name scoping topic.

The import declaration imports an external resource containing function and predicate definitions and makes it available through the given prefix.

[22] *import* ::= 'import' *uri* 'as' *ident*

Each import production causes a new tuple (prefix, 'module', uri) to be added to the prefix bindings in the query environment, where prefix is the string matching the 'ident' production, and uri the string matching the 'uri' production.

EXAMPLE 2 The declaration below would import the given module and make it available through the prefix opera.

```
import "opera.tmql" as opera
```

If the module defined the predicate inspired-by the qname opera:inspired-by would now be available.

Ed. Note.
rho thinks: What I do not understand is why this is separate from the rules/ontology stuff below. To me it seems that the agenda is: "I want to add some vocabulary/ontology knowledge (by reference or inlined) and I want to have a decent short name for this, so things are not mixed up when I use that in a query.

Examples:

```
using string for i"http://tmql.org/strings/1.0/#"
SELECT string:uppercase ("rumsti") FROM ...
# or
using adhoc for
  function rumsti (....) {
  }
```

```
  function romsti (...) {
  }

SELECT adhoc:rumsti($a) FROM ...
# or
using myrules for
  predicate influenced-by($A, $B) :- {
     pupil-of($A : pupil, $B : teacher) |
....
}.
# or whatever constraint language will come along
```

larsbot: There's a difference between importing an external module and declaring the rule/function inline. If you do the latter, most likely you only have a few, and there's no need for namespace control through prefixes. I'm not necessarily against adding the functionality you describe, just explaining the original intent. Also, I think you'll agree that declaring a prefix is different from declaring a function/rule.

6.2.2 Predicate Declarations

In addition to the predefined predicates it is possible for users to define their own predicates in TMQL, to be used for inferencing, or simply as shorthands for common subqueries. The syntax is as shown below.

[23] *rule* ::= *ident* '(' *var* (',' *var*)+ ')' ':-' *predlist* '.'

The 'rule' production adds a tuple to the rule bindings in the query environment. It is an error if the rule bindings already contain a tuple whose first element is the same as the string matching the 'ident' production. For examples, see 10.2.

Ed. Note.
Lars: Need a definition of what a rule is somewhere. Not sure where, though. Is this part of the core, or is there a middle level?

rho thinks: I would think that rules DO NOT belong into a query language. They belong to an ontology. I agree, though, that TMQL should allow to (a) import such rules/ontologies IN A DISCIPLINED WAY or (b) to define them inline INSIDE the query. TMCL would be nice if it is embeddable.

larsbot: If rules are to be definable inside queries either they must be defined in TMQL, or they must be defined in a normatively referenced standard whose grammar is merged into the TMQL grammar. At present there is no obvious candidate for the role of host to these rules, and so it seems clear that the rules belong here. Anyway, to me having this functionality matters more than which document contains the definition.

6.2.3 Function Declarations

[24] *function-declaration* ::= 'TBW'

Ed. Note.
lars says: Not much of an idea how to do this yet. Presumably functions are only used in path expressions.

rho asks: why? Would not this be a good opportunity to resolve some strange issues with 'unsafe' tolog predicates and use functions as .. functions :-?

larsbot: what I meant was that these would only be used in *value* expressions. And, yes, the idea is to get rid of the unsafe predicates, and horrible things like having to have the output of a function as a parameter to the function-as-predicate.

Ed. Note.
drrho: we could have fully-fledged functions here capable of returning all sorts of content. Only needs a page or two to do that.

larsbot: Feel free to write in a proposal. We don't have one, and we do need it.

6.3 Order Clause

All sublanguages use the same mechanism to order tuple sequences. The order clause is optional and allows the resulting tuple sequence to be sorted according to one or more given criteria:

[25] *order-clause* ::= 'ORDER' 'BY' *tuple-sequence* (',' *tuple-sequence*)+

If there are no value expressions provided in the order clause, no ordering will occur. Any number of value expressions can be provided whereby the lexical ordering is significant.

To keep path expressions concise the language allows the following abbreviations:

[C] 'asc' :== 'ASCENDING'
[D] 'desc' :== 'DESCENDING'
[E] '<<' *tuple-sequence*:== 'ORDER' 'BY' *tuple-sequence* (',' *tuple-sequence*)+
(',' *tuple-sequence*
)+ '>>'

Ed. Note.
larsbot: Again, I think having all these alternatives is just confusing and adds no benefit. Let's pick *one* syntax and stick to it. Another thing is that I can't tell where these are anchored in the syntax. Where did we define 'asc' and 'desc', and are they allowed here at all?

For every tuple in the incoming tuple sequence that tuple will be added to the current context. In this new context all value expressions in the provided tuple sequence specification are evaluated. From each result sequence one arbitrary value is taken. These are organized into an *ordering tuple*. At the end of this process every tuple of the incoming tuple sequence will be associated with exactly one order tuple.

The outgoing tuple sequence is built from the incoming tuple sequence by ordering the tuples according to the ordering tuple they are associated with.

EXAMPLE The path expression %m // composer << ./bn[0] , ./rd[* birthdate] >> will sort all composers in the map %m according to their first basename characteristics. If two composers happen to have the same name here, then the birthdate resource data is used for ordering.

6.4 Unique Clause

To filter out duplicate tuples in the incoming tuple sequence, the *unique clause* can be used:

[26] *unique-clause* ::= 'UNIQUE'

All tuples of the incoming tuple sequence are compared according to the equality rules in section Clause 5. For identical copies of tuples only one is chosen.

For further conciseness this can be abbreviated:

[F] 'uniq' :== 'UNIQUE'

Ed. Note.
larsbot: Do we have to complicate the syntax like this to save *two characters*? I think the benefit from this shortcut is non-existent, whereas the costs are high. So why bother?

6.5 Value Expressions

Ed. Note.
larsbot: I don't think the next two paragraphs belong here; they should probably be in the "types and values" clause instead.

Values are either those which have been handed in from the application into the querying process or those which are created during the execution of a query. In either case all values are *immutable*, i.e. once they exist, they

cannot be changed within a TMQL expression.

Values created within a TMQL expression are ephemeral. After creation, values can be referred to inside a given *scope* which is determined by syntactic measures as outlined below. Outside that scope the value is inaccessible.

Value expressions are expressions which produce a value, usually as a parameter to a predicate or function. They are common to all sublanguages, and have the following syntax:

[27] *value-expression* ::= *content* |
variable-reference |
function-invocation |
'(' *value-expression* ')' |
operator-expression

[28] *variable-reference* ::= *variable* ('[' *number* ']')? | *variable path-postfix* *

[29] *function-invocation* ::= *tm-reference* '(' *value-expression* (',' *value-expression*)+ ')'

Ed. Note.
larsbot: What about path expressions? I should be able to use a path expression to produce the parameters to a function, and likewise for predicates. I know the list below says this, but the grammar disagrees with it, and the grammar rules.

— By constant content according to the syntax given 5.1.

— By referencing a variable: The value is the value of the variable in the current context. If the variable is not yet bound to a value, NULL will be used.

If the variable is indexed, the number inside the [] must be a non-negative integer and is used as index. If the variable is prefixed with $ and its value is a tuple, then the index indicates the position inside the tuple which should be extracted.

If the variable is prefixed with @, then the index indicates the position of the tuple within the tuple sequence which should be extracted.

— By evaluating a function: This follows the rules in 6.2.3.

— By evaluating an operator expression.

— By evaluating a path expression: This follows the rules in Clause 7.

During the creation of an object, it will be associated with a value of one of the types listed in Clause 5. The type determines how the object can be used inside other expressions.

During the creation of an value object, it can optionally be associated with a variable:

[30] *variable* ::= ('$' | '@' | '%') (*identifier* | '_') "'*

[31] *identifier* ::= [A-Za-z0-9] [A-Za-z0-9_-]*

Variables have to be prefixed with either a $ to signal an object of simple type, @ for a objects containing sequences and % for Topic Map content. Variables can optionally be postfixed by any number of primes; these are part of the variable name. There is no limit on the length of identifiers.

Issue (_idents-are-xml-names)
Lars: Should we make idents match the XML "name" production? We need to skip the ':', obviously. There are use cases for this, but obviously also downsides.

rho: ??? Is this still an issue?

larsbot: Yes. It would be nice to not have to specify this. Also, using XML "name" would make it possible to use non-latin scripts (ie: Japanese, Arabic, Cyrillic, ...) for idents.

Some syntactic constructs allow to create nameless objects. To refer to these, the special names $_ and @_ can be used. %_ contains the current context map.

The shorthand . means the same as $_[0] and is handy if a tuple has only a single component:

[G] '.' :== '$_[0]'

EXAMPLE Valid variables are $a, $a', $_, @a_long_list_name. Examples for invalid variables are x ($ or @ is missing), $1 (identifier does not start with a character) or @list ' (no blanks before the prime is allowed).

[32] *operator-expression* ::= *value-expression operator value-expression*

[33] *operator* ::= '+' | '*' | '-' | '/' | *positive-compare-op* | *negative-compare-op*

Ed. Note.
larsbot: We need to explicitly specify wanted precedence order here.

6.6 Comments

Comments are allowed anywhere whitespace is and follows a bracket approach:

[34] *comment* ::= '/*' [^*]* '*/'

Comments cannot be nested.

Comments are ignored.

Ed. Note.
drrho: This has to be revisited later. I would more like the # comment approach, is much clearer what is commented and what not.

larsbot: Fine by me.

Ed. Note.
drrho: The above says that * cannot be inside a comment?

larsbot: Yes. Which isn't really good.

7 Path Expressions

7.1 Structure

Path expressions allow to navigate through a given topic map from a given position on. With them, values can be extracted from the context map, be they map information items or strings. As such, path expressions can extract tuples of such values, the result being then an unordered or an ordered sequence of value tuples.

A path expression is specified by a starting position, the *path base* and any number (also zero) of *path postfixes*:

[35] *path-expression* ::= *path-base* (*path-postfix* | *association-template*)*

[36] *path-base* ::= *tuple-sequence*

The *path base* defines a tuple sequence, mostly one consisting of topic map information items in the context map. Postfixes are then applied in lexical order in the context of that map. Association templates are high-level constructs allowing a concise notation for specific, but rather frequent path expression patterns.

Once the path base has been evaluated, the result will be a sequence of information items. As all postfixes operate on individual tuples or sequences of tuples of information items, the path base item sequence will be interpreted as

a sequence of singletons, each containing one information item. This sequence is called the *initial tuple sequence*. Some postfixes operate on individual tuples and some on sequences of tuples. In case of an individual tuple, postfix applications produce an *outgoing tuple sequence*. In the case of tuple sequences, postfixes transform the *incoming tuple sequence* into the *outgoing tuple sequence*.

In many cases the actual order in the outgoing sequence is not relevant for the (final) result. In this case the outgoing sequence can remain *unordered*. Otherwise the sequence will be *ordered*. All postfix applications are *stable operations*, i.e. the evaluation of every postfix occurs in the order of the incoming tuple sequence if that itself happens to be ordered. For unordered incoming tuple sequences no particular evaluation order is guaranteed.

Postfixes themselves are also always evaluated in a particular variable context. Any variable references are treated normally in that during evaluation the variable references are substituted by the variable value(s) according to the current context. Postfixes also may contain function references. These are treated normally in that during evaluation functions inside the postfix are evaluated *before* the evaluation of the postfix itself.

In the following we provide the informal semantics of path expressions. Annex D contains a formal mapping of path expressions onto the Tau-model.

7.2 Postfix Chains

Postfixes can be cascaded, so that every subsequent postfix operates on the tuple sequence which is the result of the previous postfix application. The language does not mandate specific valid combinations of postfixes. Any combination may be used whereby not all of these may make sense. A complete *cascade of postfixes* transforms the *initial tuple sequence* into a *final tuple sequence*.

A single postfix either selects topic characteristics, navigates along associations through the context map, filters specific tuples from the incoming tuple sequence or evaluates other path expressions in the context of individual tuples:

[37] *path-postfix* ::= *navigation-postfix* |
predicate-postfix |
projection-postfix

7.3 Navigation Postfix

To navigate inside a given map we have to move across associations and their roles. There are two kinds of movements, *in-movements* and *out-movements*:

[38] *navigation-postfix* ::= *navigation-in-postfix* | *navigation-out-postfix*

[39] *navigation-in-postfix* ::= '->' *path-expression*

[40] *navigation-out-postfix* = '/' *path-expression*

Navigation postfixes operate on a tuple sequence. For every tuple in the sequence the specified postfix is applied to every item in a tuple. Every such application will result in a sequence of topic map items. The result then for one particular tuple will be a new tuple of sequences. As usual, this will be flattened by creating the cartesian product of these sequences, building a sequence of tuples of topic map items. All generated tuple sequences are concatenated into the outgoing tuple sequence.

The *in-movement* is supposed to lead from a topic item to those association items where the topic plays a given role (or any direct or indirect subclass thereof). In this case the movement will return all association items in which that topic plays the given role (or any direct or indirect subclass thereof). No particular order of the output is guaranteed.

The role is specified by another, nested path expression. Its evaluation will result in a sequence of tuples as well. An evaluation error is flagged if these tuples are not singletons. The single components in these singletons are interpreted as alternative roles.

The *out-movement* is supposed to lead from an association to all topics playing (one of) the given role(s).

To retrieve the type, the scope or the reifier of an information item the following shorthand notations exists:

[H] '*' :== '->' 'instance' '[' '*' 'is-a' ']'
[I] '@' :== '->' 'statement' '[' '*' 'is-scoped-in' ']'
[J] '^' :== '->' 'topic' '\' 'reifies' '/' 'reifier'

Issue (reifier-as-assertion)
Is it topic-map correct to abuse associations for 'reification'? It is sooo much easier to see it this way instead of developing a dedicate formalism/language for 'reification'. It also would make the alignment with Tau much easier.

Issue (scope-as-assertion)
Is it topic-map correct to abuse associations for 'scope'? Scope is soooo borken (typo intended). How to deal with multiple scopes? Regard this as an abomination?

For topic items we also allow to access to the topic characteristics using the following shorthand notations:

[K] '#' 'bn' :== '->' 'topic' '[' '*' 'has-basename' ']' '/' 'basename'
[L] '#' 'oc' :== '->' 'topic' '[' '*' 'has-uri-occurrence' ']' '/' 'occurrence'
[M] '#' 'rd' :== '->' 'topic' '[' '*' 'has-data-occurrence' ']' '/' 'occurrence'
[N] '#' '*' :== '->' 'topic' '/' '*'

7.4 Predicate Postfix

With a *predicate postfix* conditions are used to filter particular tuples from the incoming tuple sequence:

[41] *predicate-postfix* ::= '[' *predicate-condition* ']'

When the predicate postfix is applied to an incoming tuple sequence the predicate condition will be evaluated for every tuple in this sequence. One by one, each of these tuples will be added to the variable context being accessible via $_ for the path expression(s) inside the predicate postfix.

Only tuples in the incoming tuple sequence where the evaluation returns a non-empty result will be part of the outgoing tuple sequence.

The language defines the following forms of conditions:

[42] *predicate-condition* ::= *path-expression*
(*positive-compare-op* | *negative-compare-op*)
path-expression
[43] *positive-compare-op* ::= '=' | '<' | '>' | '<=' | '>=' | '=~'
[44] *negative-compare-op* ::= '!=' | '!~'

In the case of two path expressions connected via a *positive compare operator* the condition is then satisfied if - after evaluating both path expressions in the current context - there exists at least one identical pair of tuples in the two result tuple sequences, respectively.

A condition can also be a single path expression:

[O] '[' ']' :== '[' '=' ']'

As both path expressions, left and right to the = comparison operator, are identical, they also result in identical tuple sequences. If that is empty, then no tuples can be compared, making the overall predicate fail. If there is a single tuple in the result, then both result sequences are identical and the predicate succeeds.

NOTE 1 These forms of conditions implement *exists semantics*: only if there exists (at least) one result, the predicate evaluates to true.

In the case of two path expressions connected via a *negative compare operator* the condition is satisified when - after evaluating both path expressions - there exists not a single identical pair of tuples in the two respective tuple sequences.

A single path expression inside a predicate can also be negated. Such predicates can be rewritten as

[P] '[' '!' *pe* ']' :== '[' *pe* '!=' *pe* ']'

If the evaluation of the path expression results in a non-empty sequence, then the comparison != between the two identical sequences will fail. Only if there is not a single tuple in the result of that path expression, the comparison will succeed emptily.

NOTE 2 The condition *"there does not exist a single combination of tuples were the comparison succeeds"* can also be rephrased as *"for all combinations the comparison will fail"*. Consequently, such predicates implement *forall semantics*.

The connective *AND* (&) between conditions we define via the following syntactic equivalence:

[Q] '[' '&' ']' :== '[' ']' '[' ']'

The connective *OR* (|) between conditions we defined based on the combination operator on tuple sequences (see 5.6):

[R] '[' '|' ']' :== '[' ']' '||' '[' ']'

To test whether a current information item has a particular value, the following shortcuts for conditions can be used conditions in predicates:

[S] '*' :== '. *' '='
[T] '@' :== '. @' '='
[U] '^' :== '. ^' '='

7.5 Projection Postfix

When operating on tuple sequences, it may sometimes be necessary to select particular columns out of the tuples in the incoming tuple sequence and to form with these new tuples for an outgoing tuple sequence. This can be achieved with a *projection postfix*:

[45] *projection-postfix* ::= '<' *tuple-sequence* (',' *tuple-sequence*)+ '>'

This postfix is applied to every individual tuple in the incoming tuple sequence. The current tuple will be added to the current variable context, addressable via the name $_. Then all path expressions inside this list of path expressions are evaluated whereby the order is left undefined.

EXAMPLE Given a map in %m the path expression %m // opera < $_[0]/bn > extracts first all operas. For every topic item a new tuple will be built which contains the sequence of basenames for that opera. The result is the sequence of all basename of all operas in the map.

Given a map in %m the path expression

```
%m // opera < ./bn[@opus], . -> work [ * is-composed-by ] >
```

extracts first all operas (using the shortcut . for $_[0]). Then a new tuple is generated for every such opera. It contains as a first component only the opus number as provided by a basename in the respective scope. The second tuple component contains one association item of type is-composed-by where that particular opera playes the role of work. If there are several composers for an opera, then for each such composer a separate tuples are created.

7.6 Association Templates

Association templates allow to define path expressions which look for associations of a certain type having particular role/player combinations. While association templates are a more concise notation for that purpose, they do not introduce new expressitivity or semantics as they can be transformed into path expressions (see Annex B). Nevertheless we will describe their informal semantics here.

Association templates can occur anywhere where a path expression postfix is allowed:

[46] *association-template* ::= *tm-reference* '(' *member* (',' *member*)+ (',' '...')? ')'

[47] *member* ::= *association-player* ':' *association-role*

[48] *association-player* ::= *path-expression*

[49] *association-role* ::= *path-expression*

The tm-reference in the association template identifies an association type and the optional members are pairs of path expressions. The ellipsis ... can be used to indicate the fact that there may be other members in such associations which remain unmentioned.

The (potential) player(s) and role(s) for a particular member in the association template are computed by evaluating the path expressions for the association player and the association role, accordingly. This happens within the current context; the evaluation order itself is irrelevant. For each member there then exists a sequence of potential players and a sequence of potential roles. If one of the sequences does not result in a sequence of singletons, an evaluation error is flagged.

The evaluation result of an association template is a singleton sequence containing all associations in the context map which satisfy the following conditions:

1. The association must be an instance of the given type or of any of its subtypes.

2. For each of the members mentioned in the association template, the association must have a role out of the sequence of potential roles, together with a player out of the sequence of potential players.

3. If the ellipsis ... has not been used, there does not exist further members in the association which are not explicitely named in the association predicate.

EXAMPLE The following association template identifies all cities which contain theatres:

```
is-located-in (%_//cities: location , $_: theatre)
```

The path expression %_//cities selects only the cities in the current context map, only those will be considered. The variable $_ acts here as wildcard, as we do not care to memorize and post-process the theatre itself here.

8 SELECT Expressions

8.1 Structure

SELECT expressions use a pattern-matching approach for queries. They always produce sequences of tuples as their results. A query follows the syntax:

[50] *select-expression* ::= *select-full* | *predlist*

[51] *select-full* ::= *select-clause ?*
from-clause ?
where-clause2

[52] *where-clause2* ::= 'WHERE' *predlist*

Full SELECT expressions are evaluated starting with the FROM clause, which produces the topic map which is queried. Then, the predicate list in the WHERE clause is evaluated, generating an initial tuple sequence. Finally, the SELECT clause is evaluated, producing the final tuple sequence.

If only a predicate list is given, this is equivalent to having a full SELECT with no SELECT or FROM clause.

In the following we provide the informal semantics for the individual clauses. A formal mapping to FLWR expressions is given in Annex A.

8.2 FROM Clause

[53] *from-clause* ::= 'FROM' *tm-content*

The optional FROM clause may contain references to one or more maps which the query refers to, or an expression computing the map. If the FROM clause is omitted, this is equivalent to FROM %_.

Ed. Note.
larsbot: The %_ syntax is disgusting. We should find something better.

8.3 SELECT Clause

SELECT clauses are used to define which values go into the final result set using references to variables defined in the query or expressions based on these.

[54] *select-clause* ::= 'SELECT' *value-expression* (',' *value-expression*)+

The SELECT clause generates an output tuple for every input tuple in the tuple sequence, with one component in the output tuple for each value-expression in the SELECT clause in lexical order.

EXAMPLE The query below lists all composers which have composed an opera. The $OPERA variable is only used internally in the query, and does not show up in the final query result.

```
select $COMPOSER
where composed-by($COMPOSER : composer, $OPERA : work)
```

8.4 Predicate lists

8.4.1 General

[55] *predlist* ::= *predexpr* (',' *predexpr*)+

The predicate list produces a tuple sequence, with one component for each variable occurring in the predicate list, and one tuple for each binding of values to the set of variables. Each tuple represents a set of values for the variables in which the statement represented by the query is true, according to the topic map(s) being queried.

Variables occurring within a predicate list are scoped for the whole predicate list. This implies that two occurrences of variables with the same name refer to one and the same variable binding.

The predicate expressions are evaluated one by one with the result set produced by the previous expression as input to the next. The initial result set used by the first expression has one column for each variable in the query and a single row where all values are null. The result set returned by the final expression is the result of evaluating the predicate list.

Predicate expressions have the following syntax:

[56] *predexpr* ::= *predicate* |
compexpr |
instance_of |
notpred |
orpred |
optionalpred |
for_every

8.4.2 Predicate application

The default form of predicate expression is the application of a predicate, which is done with the following syntax:

[57] *predicate* ::= *tm-reference* '(' *member2* (',' *member2*)+ ')' ('as' *variable*)?

[58] *member2* ::= *variable* | *simple-content* | *tm-reference* | *pair*

[59] *pair* ::= *value-expression* ':' *tm-reference*

The type, number, and order of parameters is defined by the predicate that is applied. Some predicates are sensitive to the order of their parameters, while others are not.

Ed. Note.
Lars: Need to describe how predicates are looked up. Also need to describe how predicates are evaluated and all that. Leaving this for later. Also need to consider whether to allow path expressions here.

8.4.3 Comparisons

Comparison expressions are used to compare values produced by means of path expressions, which often are simple variable references. Comparison expressions have the following syntax:

[60] *compexpr* ::= *value-expression positive-compare-op* | *negative-compare-op value-expression*

Comparison operations are mapped to predicate applications where the comparator is the predicate, and the value expression are parameters in the order given.

8.4.4 The instance-of predicate

The instance-of predicate has special syntax, as follows:

[61] *compexpr* ::= *value-expression* ':' *value-expression*

Any expression matching the production above can be transformed into an application of the predicate instance-of with the parameters in the same order.

8.4.5 OR Expressions

Or expressions are used in queries where a result is wanted when it is sufficient for one of several conditions to be met. The syntax is as follows:

[62] *orpred* ::= '{' *predlist* '|' *predlist* + '}'

Ed. Note.
rho: should not there be a () around the |-predlist group?

larsbot: That's not necessary.

The predicate lists in the OR expression are evaluated independently, all starting from the input result set. The result of the or expression is the concatenation of the output result set from all the predicate lists in the expression.

8.4.6 NOT Expressions

NOT expressions are used to remove from query results all matches where a certain condition is met, and the syntax is as follows:

[63] *notpred* ::= 'not' '(' *predlist* ')'

The NOT expression is evaluated by evaluating the predicate list with the current input result set as the input result set. The output from the NOT expression is a result set containing all rows in the input result set that do not match a row in the output result set from the predicate list. A row matches another if for all columns where neither has a null value their values are equal.

Ed. Note.
rho: we need consistent terminology: set or sequences?

8.4.7 Optional clause

The optional clause is used to include predicate expressions in the query that do not form part of the search criteria (ie, they do not remove matches), but which are included to produce values (ie, they bind variables). Optional clauses have the following syntax:

[64] *optionalpred* ::= '{' *predlist* '}'

Ed. Note.
This may not make it through to the final language, since we can now (mostly) do this by means of path expressions in the SELECT part.

Queries using optional clauses can always be rewritten into a longer, but equivalent query that does not use the optional clause. An optional clause with predicate list X can be replaced by the following OR expression: { X | 0 = 0 }. That is, an OR expression where X is one branch, and the other branch is trivially true.

8.4.8 EVERY/SATISFIES Expressions

In general, predicate expressions generate as matches all cases where at least one piece of evidence can be found to support the match. EVERY/SATISFIES expressions do the opposite: they only return matches in cases where for *all* values in a certain set there is supporting evidence.

[65] *for_every* ::= 'EVERY' *predlist* 'SATISFIES' *predlist*

All EVERY/SATISFIES expressions can be translated into an equivalent (but rather harder to understand) nested NOT expression, as follows. The expression EVERY x SATISFIES y can always be replaced with not(x, not(y)), which will give the same result.

EXAMPLE An example of the use of EVERY/SATISFIES expressions would be the following:

```
$CLUSTER : cluster,
EVERY part-of($MACHINE : part, $CLUSTER : whole)
SATISFIES is-down($MACHINE)?
```

The rationale for this query is that clusters are sets of redundant hardware components providing fail-over for each other; this query finds all clusters which are down, that is, clusters where every part of the cluster is down.

9 FLWR Expressions

9.1 Structure

FLWR expressions have the same expressivity as path expressions; their syntactic structure, though, allows not only tuple sequences to be returned, but also the construction of content in XML and TM form:

[66] *flwr-expression* ::= (*for-clause* | *let-clause*)*
where-clause ?
return-clause

Only the *RETURN clause* is obligatory; it allows the result content to be generated. All the other clauses are optional.

The conceptual processing model of FLWR expressions is that first the *FOR clause*s are evaluated in lexical order, if they exist. A value expression is used to generate a tuple sequence. One by one, each of these tuples will be bound to the loop variable which is named in the *FOR clause*. This variable binding will be added temporarily to the current context while the rest of the query is evaluated. This step is repeated for every tuple in the sequence. If

there is more than one such *FOR clause*, these act consequently as nested loops.

The optional *WHERE clause* contexts to be tested for particular conditions; only those contexts for which the evaluation of the WHERE clause returns TRUE will be passed on into the next clause.

Note that a WHERE clause can contain references to variables that are not bound to values in the current context. Conceptually, for each such variable another *FOR clause* has to be created where the variable iterates over all items in the current context map.

Ed. Note.
larsbot: "items" is a bit weak. We should be able to be more specific. (Similarly, "context" is handy, but also perhaps weaker than it should be.)

Once the variable contexts have been filtered, all these remaining contexts are now organized into an unordered sequence.

The *RETURN clause* is then responsible for actually generating content, be it a number, a string, a tuple sequence, a single XML structure, or a TM instance.

If—due to enclosing *FOR clauses*—several results have been computed for every iteration, then these partial results are combined into a total result. For strings this combination is done via concatenation, for tuple sequences, the partial sequences are interleaved (unless they are ordered), for XML content the individual fragments are combined into a node list and for Topic Map content the fragments are merged.

In the following we informally describe the semantics of the individual clauses. A formal mapping of FLWR expressions to path expressions is given in Annex C.

9.2 FOR Clauses

A given FLWR expression can contain any number of *FOR clauses*. In each of these clauses a *loop variable* is declared:

[67] *for-clause* ::= 'FOR' *variable-association* (',' *variable-association*)+

The language allows several such variable associations to be listed in a single *FOR clause*. This is equivalent to using a dedicated FOR clause for every individual variable.

A variable association includes the evaluation of another query expression:

[68] *variable-association* ::= *variable* 'IN' *value-expression*

The introduced variable is visible until the end of the directly enclosing FLWR expression.

The provided value expression is evaluated in the current context, resulting in a tuple sequence. If the variable can only hold simple content, then - one by one - one tuple of this sequence is bound to the variable and this binding is added to the current context. For each of the tuples the rest of the FLWR expression is evaluated.

If the variable can hold composite content, then the whole tuple sequence is assigned and that binding is added to the current context. In that case, only a single iteration of the remained of the FLWR expression is evaluated.

9.3 LET Clauses

LET clauses are used to assign a value to a variable in the current context without looping over all alternatives. The syntax is as given below.

[69] *let-clause* ::= 'LET' *variable* ':=' *value-expression*

The value expression will be evaluated in the current context. If the variable is prefixed with @ the resulting tuple sequence will be bound as a whole to the named variable. If the variable is prefixed with $ only the first value out of

this sequence will be bound to the variable. If the result tuple sequence is empty, then NULL will be used as value. That variable binding will be added to the current context.

Ed. Note.
larsbot: The $ and @ prefixes don't seem like a very good idea. In any case, if they are to be generally supported throughout TMQL they should not be introduced here.

A LET clause where the variable has a $ prefix is equivalent to a FOR clause with the same value-expression, except that it must have [0] appended. If the variable has a @ prefix it is equivalent to a FOR clause with the same value-expression.

Ed. Note.
larsbot: Are these equivalences actually correct?

9.4 WHERE Clause

9.4.1 Structure

The *WHERE clause* makes use of boolean expressions but expresses *exists semantics*:

[70] *where-clause* ::= 'WHERE' *exists-clause*

[71] *bool-expression* ::= *bool-expression* 'AND' *bool-expression* |
bool-expression 'OR' *bool-expression* |
'NOT' *bool-expression* |
'(' *bool-expression* ')' |
predicate-condition |
forall-clause |
exists-clause

As convenience, it is possible to reduce the verboseness when dealing with path expressions:

[V] 'WHERE' :== 'WHERE' 'EXISTS'

All boolean expressions are evaluated in the current context.

For the evaluation, the operators AND, OR and NOT have the usual meaning (non-short-circuit, symmetric), the brackets () can be used to override the usual precedence, namely that AND binds stronger than OR and that NOT binds stronger than AND.

EXAMPLE As usual, the operators AND and OR bind the *immediate* boolean expressions. In the example

```
EVERY $opera IN %m // opera SATISFIES
    composed-by ($person: composer, $opera: opera)
    AND
    SOME ...
```

the AND binds the composed-by and the SOME clause as the nesting suggests and not as the following indentation insinuates:

```
EVERY $opera IN %m // opera SATISFIES
    composed-by ($person: composer, $opera: opera)
AND
SOME ...
```

Issue (not-considered-harmful)
NOT can pose problems here? Can we reduced it with forall/exists?

All subexpressions of a boolean expression are interpreted in the current context. They evaluate either to TRUE or

FALSE.

If boolean expressions in a WHERE clause contain references to unbound variables, then the directly enclosing FLWR expression will be wrapped with an additional *FOR clause* which makes the variable iterate over all items in the current context map. With this process all unbound variables are eliminated.

The evaluation of boolean expression works under the following constraints. For any context which satisfies a boolean expression, it must hold that:

1. Any two occurrences of one and the same variable within the boolean expression must have equivalent values bound to them in that context.

2. Any two occurrences of two different variables within the boolean expression may have any values bound to them. They are regarded to be independent.

3. Any two occurrences of two different variables within the boolean expression where the names only differ in the number of primes, must not have equivalent values bound in that context.

The special variables $_ and @_ can be used as placeholders, but any bound values are not accessible outside the directly enclosing WHERE clause.

9.4.2 FORALL and EXISTS Clauses

These clauses allow to test whether a particular condition can be satisfied by at least one value out of a set of values or whether all these values satisfy a condition:

[72] *forall-clause* ::= 'EVERY' *variable-association* (',' *variable-association*)+ 'SATISFIES' *bool-expression*

[73] *exists-clause* ::= 'SOME' *variable-association* (',' *variable-association*)+ 'SATISFY' *bool-expression*

Any number of new variable associations can be introduced. These bindings will be added to the current context only for the evaluation of the boolean condition.

The *forall clause* evaluates to TRUE if for all variable bindings the boolean expression evaluates to TRUE. Otherwise it will evaluate to FALSE. The *exists clause* evaluates exactly then to TRUE if there exists at least one variable binding for which the evaluation of the boolean clause returns TRUE.

If the boolean condition is not relevant, the following abridged forms can be used:

[W] 'EXISTS' *p* :== 'SOME' 'IN' 'SATISFY' 'TRUE'
[X] 'NOT' 'EXISTS' *p* :== 'NOT' 'EVERY' 'IN' 'SATISFIES' 'TRUE'

In the special case that two path expressions are compared for equality, then we also introduce

[Y] 'EXISTS' '=' :== 'SOME' 'IN' ',' 'IN' 'SATISFY' '='
[Z] 'NOT' 'EXISTS' '=' :== 'EVERY' 'IN' ',' 'IN' 'SATISFIES' '!='

9.5 RETURN Clause

In the RETURN clause the variable bindings in the current context can be used to generate output:

[74] *return-clause* ::= 'RETURN' *value-expression*

For every context generated by the previous clauses of the FLWR expression, the *RETURN clause* is evaluated. The aggregated result of this evaluation is the result for the directly enclosing FLWR expression.

Ed. Note.
rho: Somehow I think, we need an if-then-else

10 Predefined Environment

10.1 General

Ed. Note.
bla, bla environment automatically imports the package STANDARD and associates the empty prefix with it.

for strings:

- concat, length, the usual suspects
- pattern match =~, !~
  Issue: how to fit in patterns for =~ qr{...} as shorthand for re:match (..., ..., 'i'), argh.

for sequences:

- || interleave, ..., &, |

other topics

- is-scoped-by
- is-reified-by
- has-uri-occurrence
- has-data-occurrence
- has-basename

-topic?

-thing?

dates:?

10.2 Predefined Predicates

The following predicates are predefined in TMQL.

```
using xtm for i"http://www.topicmaps.org/xtm/1.0/core.xtm#"

direct-instance-of($INSTANCE, $TYPE) :-
  xtm:class-instance($INSTANCE : xtm:instance, $TYPE : xtm:class).

is-subclass-of($SUB, $SUPER) :- {
  xtm:superclass-subclass($SUB : xtm:subclass, $SUPER : xtm:superclass) |
  xtm:superclass-subclass($SUB : xtm:subclass, $MID : xtm:superclass),
  is-subclass-of($MID, $SUPER)
}

instance-of($INSTANCE, $TYPE) :- {
  direct-instance-of($INSTANCE, $TYPE) |
  direct-instance-of($INSTANCE, $SUB),
  is-subclass-of($SUB, $TYPE)
}.
```

11 Conformance

Ed. Note.

Generally: TMQL defined as a function of two parameters: environment + query; implementations conform if they return the result required by this specification. We may need to add language to allow implementations to have facilities for restricting resource usage and DoS-preventive meeasures. General approach: whatever we don't specify is up to you.

Ed. Note.
rho: should resource limits (time x space) be mentioned in this section? "A processor might be conformant, even if it cancels processing ....".

larsbot: It would be a strange standard where this wasn't implicit, but I'm happy for it to be spelled out here once and for all. Repeating this throughout the spec in various forms I do not think is acceptable.

Annex A
(normative)

Mapping of SELECT Expressions to FLWR Expressions

SELECT expressions can be mapped to FLWR expressions. This is quite intuitive, since SELECT expressions solve the special case of generating tuples out of a particular map. Both use the same concept of a boolean expression for the condition and both use the same sorting mechanism:

[AA] 'SELECT' ',' ';;;;;'
'
'FROM'
'WHERE'
'FOR' 'IN' '// *'
'WHERE'
'RETURN' '(' ',' ', ..., ' ')'

If an *ORDER clause* exists, that would be copied verbatim, so would be a *UNIQUE clause*.

The path expressions would be converted according to the following pattern:

— Every anonymous object in *OD* corresponds its path expression in *OD'*.

— Every typed object in *OD* corresponds to an additional LET clause in the FLWR expression of the form:

LET *variable* := *tau* // *tm-reference*

This, new, variable will appear in *OD'*.

— Every untyped object in *OD* corresponds to a LET clause in the FLWR expression of the form:

LET *variable* := *tau* // *

The variable will appear in *OD'*.

Ed. Note.
larsbot: What about the WHERE clause?

Annex B
(normative)

Mapping Association Templates to Path Expressions

TBW

Annex C
(normative)

Mapping of FLWR Expressions to Path Expressions

FLWR expressions consist of two parts. In the *upstream-side* variables will be bound to values to build contexts. These contexts are optionally sorted and finally filtered with the WHERE clause. The *downstream-side* of a FLWR expression is the RETURN clause which is concerned in translating the variable bindings into content, be it tabular, XML content or Topic Map content.

The upstream part of FLWR expressions can be mapped into path expressions as follows:

1. The tau expression for the queried map %m is used as absolute path base %m // *.

2. Then a projection postfix is built by including all path expressions (keeping the syntactic order):

   [AB] 'FOR' ':=' :== '<' ',' ',' '...' ',' ',' ',' '...' '>'
   'FOR' ':='
   '...'
   'LET' ':='
   'LET' ':='
   '...'

   The variable names , , ..., , , ... are associated with an index in the projection tuple. is associated to 0, to 1, etc. These associations can be used in the RETURN clause to refer to individual values.

Ed. Note.
I have a problem with LET clauses and variables which store lists. I cannot hold lists in a tuple component.

3. The condition in the WHERE clause is converted into predicate postfixes as follows:

1. All boolean subexpressions which are separated by AND are converted into a separate predicate postfix.

2. All boolean subexpressions which are separated by OR are put into one predicate postfix.

3. TODO: FORALL, EXISTS

4. TODO: association-predicate becomes path expression

4. The ORDER clause is added as further postfix.

5. The UNIQUE clause is added as further postfix.

Annex D
(normative)

Path to tau Expression Translation

Ed. Note.
Robert to put Tau model here, plus TMDM representation in it.

Annex E
(informative)

tau Model

TBW

Annex F
(normative)

Relationship between Tau and TMDM

The Topic Map Data Model (TMDM) describes a data model for Topic Maps. In that it defines which objects exist in a given Topic Map instance and what properties these different objects have. In contrast, the Tau model sees a topic map instances as a set of assertions; all other features of a topic map are mapped into assertions.

In the following we will characterize how topic map instances according to TMDM can be thought as instances of a map according to Tau

need more time for typing....

Annex G
(informative)

Syntax

Core syntax and commodity syntax will be inserted via XSLT.

Bibliography

[1] *TMQL Requirements*, ISO, 2003

[2] *TMQL Use Cases*, ISO, 2003