Topic Maps Query Language

2006-05-02

Lars Marius Garshol, Robert Barta

Contents

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.

Topic Maps Query Language

1 Scope

This International Standard defines a formal language for accessing information organized according to the Topic Maps paradigm. This is achieved 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 between a querying application and a query processor, 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 functionality every conformant processor must provide.

This International Standard does not define an API (application programming interface) to interact with query processors nor does it define an extension mechanism for extending the predefined environment. It does provide means for importing external ontologies and functionality. This International Standard also remains silent on other implementation issues, such as optimization or error recovery.

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, http://www.isotopicmaps.org/sam/sam-model/

TMRM, *ISO 13250-5 Topic Maps — Reference Model*, http://www.isotopicmaps.org/tmrm/

CTM, *ISO 13250-? Topic Maps — Compact Notation*, ?????

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

XSDT, *XML Schema Part 2: Datatypes Second Edition*, W3C, W3C Recommendation, 28 October 2004, http://www.w3.org/TR/xmlschema-2/

RFC3986, *RFC 3986 - Uniform Resource Identifiers (URI): Generic Syntax*, The Internet Society, 2005, http://www.ietf.org/rfc/rfc3986

RegExp????, *Something which contains RegExps*,

3 Notation

3.1 Syntax Conventions

The syntax of TMQL is defined on three levels. At the *token level* we make use of regular expressions [RegExp????] to specify patterns for valid text tokens. If tokens are defined outside the grammer, a reference is provided.

The *canonical level* is defined using context-free grammars (EBNF, [XML 1.0]). The productions are numbered for reference.

On top of the canonical syntax, shorthand notations are introduced to reduce the syntactic noise in actual query expressions. These abbreviations are defined via an additional grammar production whereby a term (a sequence of terminals and non-terminals) in the right-hand side of the production is mapped to a second, more canonical term. Any abbreviated form and its canonical form are semantically equivalent. These mappings are numbered with letters for reference.

As usual, terminal symbols are either delimiting or not. The list of delimiting symbols is provided in Annex A. Any other symbol is not delimiting and whitespace characters (blank, tab and newlines) must be used for separation. Whitespace characters are allowed everywhere between two terminals; as usual, this is not encoded explicitely in the syntax.

Annex B contains the complete language syntax. Please note that the grammar was produced for human consumption, and is not optimized for a particular technology, sentential structure (LL(k) or LALR) or for a minimum of non-terminals.

Ed. Note:
Everything which contains @@@@@@ is still in limbo. Everything else claims not to be :-)

Ed. Note:
For a compact TM notation we use here AsTMa= 2.0 AsTMa[3] until CTM has emerged. At least we see what the requirements from TMQL are.

3.2 Ontological Comittments

This International Standard makes the following prefix references to external vocabulary:

xtm
http://www.topicmaps.org/xtm/

This is the namespace for the concepts used by XTM.

xsd
http://www.w3.org/2001/XMLSchema

This is the namespace for the XML Schema Datatypes.

tmql
http://www.topicmaps.org/tmql/1.0/

Under this prefix the concepts of TMQL itself are located.

fn
http://www.topicmaps.org/tmql/1.0/functions

Under this prefix user-callable function of the predefined TMQL environment are located.

op
http://www.topicmaps.org/tmql/1.0/operators

Under this prefix unary and binary operators of the predefined TMQL environment are located.

3.3 Informal and Formal Semantics

The semantics of TMQL is defined in prose and also formally. The prose semantics is highlighted in the text using a vertical sidebar; it—by its nature—is ambiguous and is only meant to support the human reader.

The informal semantics is superceded by the formal semantics. The *static semantics* maps every canonical syntactic form onto TMRM path expressions [TMRM]. The *dynamic semantics* is given by the evaluation function for TMRM path expressions which operate on TMRM instance data. The semantics for TMDM instances follows indirectly via the mapping of TMDM instances into TMRM instances 9.2.

Ed. Note:
Say something about the notation here?

TMQL is semantically neutral, with the exception of *class transitivity*. This implies that if a concept B is a superclass of A, and C is a superclass of B, then also C is a superclass of A. This also implies that if a concept is an instance of A, that very concept is also an instance of B (and C).

4 Content

4.1 General

In the following, the term *content* is used to characterize structured information, be it statically denoted or dynamically generated. Such information can be of simple type, such as numbers and strings; or such information can be structured, as in sequences of tuples (tables), it can XML-based or it can be organized along the Topic Map paradigm.

[0]	*content*	→	*simple-content* |
			(*tuple-expression* **)** |
			""" *tm-content* **"""** |
			xml-content |

			{ *query-expression* **}**
[1]	*simple-content*	→	(*atom* | *item-reference* | *variable*) [*navigation*]

Content is either a reference into the context map (5.4), or an atom which refers to a constant; or a variable which—after evaluation in the current context (Clause 5)—will render a value. Any such value can be the starting point of a *series navigational steps* inside the contextual map.

4.2 Atoms

Atoms are constants, such as strings, dates or integers, and URIs. This International Standard recognizes natively a small set of primitive data types (see Clause 7) together with operators on objects of these types.

[2]	*atom*	→	**null** | *boolean* | *integer* | *decimal* | *uri* | *date* | *string* [**^^** *qname*]
[3]	*boolean*	→	**True** | **False**
[4]	*integer*	→	`[0-9]+`
[5]	*decimal*	→	`[ - ] [0-9]+ [ . [0-9]+ ]`
[6]	*date*	→	**...xsd:dateTime...**
[7]	*uri*	→	**...xsd:anyURI...**
[8]	*string*	→	`" { [^"] } "`

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.

Integers, decimal numbers, dates, URIs and strings themselves are directly recognized by the language. For all other types, a URI can be provided to indicate the data type. For every data type provided, it is assumed that implementations specify deserialisation and serialisation rules between values of this type and their string representation(s); they also have to define a boolean operator to compare for equality and define ordering on values of this type.

EXAMPLE:
The following are valid atoms:

```
"23"^^http://www.w3.org/2001/XMLSchema#integer
23
"Hello World"
3.1415
2005-10-16T10:29
http://example.org/something
```

The following are invalid atoms:

```
3,14        #  (comma instead of dot)
- 273.15    # (no blanks between the sign and the value are allowed).
```

4.3 Item References

Items in the context map can be identified by either their *source locator* (internal item identifier), a *subject address* (if such exists) or a *subject identifier* (which can be used for indicating the subject).

[9]	*item-reference*	→	*identifier* | *qname*
[10]	*qname*	→	*uri* | *prefix* *identifier*
[11]	*prefix*	→	**/\w+:/**
[12]	*identifier*	→	**/\w[\w\-\.]*/**

If the item reference is an identifier (without prefix) then this identifier is interpreted as a *source locator* for a topic in the context map. The result is then this topic item. If no such topic exists, an error will be flagged (`????`).

EXAMPLE:
The following content first identifies the topic with the internal identifier `jack` and then retrieves all names:

```
jack / name
```

If the item reference is an absolute URI, then that is used for further processing (4.4).

EXAMPLE:
The following content is interpreted first as a URI value. The trailing `=` then signals that the URI should be interpreted as source locator for a topic in the context map.

```
http://example.org/something/ =
```

If the identifier has a prefix, then that prefix (without the trailing colon `:`) must exist as source locator for a topic of type `tmql:ontology` in the environmental map (Clause 7); it is an error (`???`), if no such topic exists. If a subject indicator for that ontology exists, then—together with the identifier— an absolute URI according the rules [RFC3986] (5.2, Relative Resolution) is constructed. That URI is interpreted as *subject indicator* for a topic in the context map. If a subject locator for that ontology exists, then —together with the identifier— an absolute URI according the rules [RFC3986] (5.2, Relative Resolution) is constructed. That URI is interpreted as *subject locator* for a topic in the context map. If no such topics exist, an error will be flagged (`????`).

EXAMPLE:
The content `xtm:subject` is recognized as the topic for the XTM concept *subject*, as `xtm` is a predefined prefix for an ontology covering XTM concepts (Clause 7).

Additional Notation: The `*` always stands for the *thing*, i.e. it indicates `xtm:subject`.

[A]	*item-reference*	→	*
		:==	**xtm:subject**

4.4 Navigation

Each navigation step occurs along one axis within a Topic Map structure [TMDM]. A step can be either followed in *forward* (`:>:`) or in *backward* (`:<:`) direction:

[14]	*navigation*	→	*step* [*navigation*]
[15]	*step*	→	*axis* (**:>:** | **:<:**) [*item-reference*]
[16]	*axis*	→	**classes** |
			superclasses |
			players |
			roles |
			characteristics |
			scope |
			reifies |
			address |
			indicator |
			atomify

The item reference adds typing information which is useful with some axes. If it is missing, the default `xtm:subject` will be assumed.

Given a context map and a single value (be it an atom or an item in the context map), the following axes are defined. All combinations not mentioned below render empty results:

`classes`

In forward direction this step computes all *classes* (types) of the value. In backward direction this step produces all *instances* of the value. The optional item identifier has no relevance.

EXAMPLE:
`person classes :<:` produces all instances of the concept `person`, say, `jack`, `jill`, etc.

Additional Notation: Asking for all classes of an item is the inverse of asking for instances. Also retrieving superclasses can be interpreted as an inverse of retrieving subclasses.

[B]	*step*	→	**instances**	**:>:**	*item-reference*
		:==	**classes**	**:<:**	*item-reference*
[C]	*step*	→	**subclasses**	**:>:**	*item-reference*
		:==	**superclasses**	**:<:**	*item-reference*

`superclasses`

In forward direction this step computes all *superclasses* of the value. In backward direction this step produces all *subclasses* of the value. The optional item identifier has no relevance.

EXAMPLE:
`person superclasses :>:` produces all superclasses of the concept `person`, say, `human`, `mammal`, etc. depending on the used upper ontology.

`players`

If the value is an association item, in forward direction this step computes all *role-playing items* of that item. The optional item identifier specifies the type of the roles to be considered whereby class transitivity is respected. If a playing topic plays different roles in such an association item, then it may appear several times in the result.

If the value is a topic item, in backward direction this step computes all association items in which that topic plays a role. If a role playing topic appears under several roles in one and the same association, this assocation will appear several times. The optional item identifier specifies the type of the roles to be considered whereby class transitivity is respected.

EXAMPLE:

The following navigation finds first all associations where a topic item (bound to `$p`) is playing the role `member`. Then for all these the players for role `group` are retrieved.

```
$p players :<: member players :>: group
```

Additional Notation: Following shorthand notations for movements involving association items are available:

[D]	*step*	→	**->**	*item-reference*
		:==	**players** **:>:**	*item-reference*
[E]	*step*	→	**<-**	*item-reference*
		:==	**players** **:<:**	*item-reference*

EXAMPLE:
The following navigation chain finds first all associations where a topic item (bound to `$p`) is playing the role `member`. Then for all these the players for role `group` are retrieved.

```
$p <- member -> group
```

roles

If the value is an association item, in forward direction this step computes all *role-typing* topics. Multiple use of the same role cause multiple results.

If the value is a topic item, in backward direction this step computes all association items where that topic is a role type. Multiple uses of one topic in one association cause multiple results.

The optional item identifier has no relevance.

EXAMPLE:
The following navigation finds first all involvements of `jack` as a `member` and then all roles in these associations:

```
jack players :<: member roles :>:
```

characteristics

If the value is a topic item, in forward direction this step computes all characteristics of that topic. The optional item identifier specifies the type of these characteristics whereby class transitivity is honored. is used to select those characteristics which are direct or indirect subclasses thereof. The result is a sequence of characteristic items (not atomic values).

If the value is a characteristic item, in backward direction this step computes the topic to which the characteristic is attached. The optional item identifier can be used to control the type of the characteristics one is interested in.

EXAMPLE:
The following navigation finds all `homepage` characteristics of `jack`:

```
jack characteristics :>: homepage
```

EXAMPLE:
To retrieve all characteristics one can use `jack characteristics :>: *`.

EXAMPLE:
The following navigation chain finds all characteristics of a topic item stored in `$t` and then extracts all types from these:

```
$t characteristics :>: * classes :>:
```

scope

In forward direction, the movement leads from characteristics and association items to their scope.

If the value is a topic, in backward direction this movement leads to all associations and characteristic items in that scope. The optional item reference can be used to select the type of these, to be considered.

Additional Notation: To extract scoping information the following shorthands exist:

[F]	*step*	→	**@**	
		:==	**scope** **:>:**	*

address

If the value is a topic item, in forward direction this step retrieves all subject addresses of this item.

If the value is an absolute URI, in backward direction this step retrieves all topic items which have this URI as subject address. It is an error (`????`) if no such topic exists in the context map. The optional item identifier has no relevance.

Additional Notation: For identifying topics via a subject address, the following shortcut is introduced:

[G]	*step*	→	**=**

:== **address** **:<:** *

EXAMPLE:
If an XTM instance would be stored in a file `file:/maps/dictators.xtm`, then the expression `file:/maps/dictators.xtm =` would identify the document itself.

`indicator`

If the value is a topic item, in forward direction this step retrieves all subject indicators of this item. If the value is an absolute URI, in backward direction this step produces the topic which has this URI registered as subject indicator. The optional item identifier has no relevance.

Additional Notation: For identifying topics via a subject identifier, the following shortcut is introduced:

[H] *step* → **~**
:== **indicator** **:<:** *

EXAMPLE:
In a map about dictators the expression `http://en.wikipedia.org/wiki/Stalin ~` would indicate the subject 'Stalin'.

`reifies`

If the value is a topic item, then in forward direction this steps finds the association or characteristics which is reified by this topic. If the topic reifies a map, then all items in the map will be returned and the context map `%_` will be set to this for the remainder of path expression in which this step occurs.

If the value is an association or a characteristic item, then in backward direction this steps finds the reifying topic.

Additional Notation: To zoom into an association, characteristics or a whole map the following shorthands exist:

[I] *step* → **>>**
:== **reifies** **:>:** *

`atomify`

If the value is a characteristic item, in forward direction this step *schedules* the item for *atomification*, i.e. marks the item to be converted to the atomic value (integer, string, etc.) within the characteristic. The item is effectively converted to an atom according to the atomification rules (4.5). The optional item identifier has no relevance.

If the value is an atom, in backward direction this step *de-atomifies* immediately the atom and returns all characteristics where this atom is used as value. The optional item identifier is used to select those characteristics which are direct or indirect subclasses thereof.

EXAMPLE:
The following navigation finds all `homepage` URLs of `jack`:

```
jack characteristics :>: homepage :>: atomify
```

EXAMPLE:
The following navigation finds all topics which have a `homepage` characteristic given a certain URL:

```
"http://myhomepages/jack"^^xsd:anyURI :<: atomify * characteristics :<: homepage
```

Additional Notation: If topic characteristics should be automatically atomified the following shorthand can be used at the end of a navigation chain:

[J] *navigation* → **/** *item-reference*
:== **characteristics** **:>:** *item-reference* **atomify** **:>:** *

EXAMPLE:
To extract all values of the characteristics of type `homepage` from a topic item bound to `$p`, one would write:

```
$p / homepage
```

Additional Notation: If atoms should be automatically converted into characteristics items of a certain type, the following shorthand can be used:

[K] *navigation* → **** *item-reference* [*navigation*]
:== **atomify** **:<:** * **characteristics** **:<:** *item-reference* [*navigation*]

EXAMPLE:
The following computes all topic items where the integer `23` is used as value in an occurrence of type `age`:

```
   23 \ age
```

Whenever a typing identifier is used, then class transitivity is honored.

EXAMPLE:
In the following expression not only `homepage` occurrences are extracted but also all those characteristics which are also instances of subclasses of `homepage`.

```
jack characteristics :>: * homepage
```

If a navigation step is applied to a sequence of values, it is applied to all values and the results are combined. No ordering in these sequences is guaranteed.

NOTE:
There is no navigation axis for deriving the data type of an atom.

4.5 Auto-Atomification

Topic characteristics are experienced in an ambivalent way: either as characteristics item, including not only the value, but also the scope and the type of the characteristic; or as the atomic value alone.

When a characteristic item is subjected to an *atomify* navigation step, the atomic value is not immediately extracted, but the process is postponed.

EXAMPLE:
The following query expression will return only those homepage URLs where the `homepage` characteristic is in scope `wikipedia`:

```
select $p / homepage [ @ wikipedia ]
where
   $p isa person
```

If the atomification would immediately be done, the information for scope (and type) would be lost for further processing steps. Instead, this step is postponed until one of the following situations occur:

1. The characteristics is about to be passed to the environment as part of the result.
   EXAMPLE:
   The following query expression will return the homepage URLs as URIs:
   ```
   select $p / homepage
   where
      $p isa person
   ```
2. The characteristics is about to be compared to an atomic value.
   EXAMPLE:
   The following query expression will return all persons younger than 18:
   ```
   select $p
   where
      $p / age < 18
   ```
3. The characteristics is about to be compared in the process of ordering (`asc` or `desc`).
   EXAMPLE:
   The following query expression will return person name(s) and age(s), in age descending order:
   ```
   select $p / name, $p /age
   where
      $p isa person
   order by $p / age desc
   ```
4. The characteristics is about to be passed into a function in the process of a function invocation.
   EXAMPLE:
   The following query expression will return person name(s) and their age which is computed from the birthday using a custom function `age`:
   ```
   select $p / name, age ($p / birthday)
   where
      $p isa person
   ```
5. The characteristics is about to be passed into a predicate in the process of predicate invocation.
   EXAMPLE:
   The following query expression will return all persons younger than `methusalem`:
   ```
   select $p
   where
      is-younger ($p, methusalem / age) &
      $p isa person
   ```
6. The characteristics is used inside a TM fragments, except when used on positions where a characteristic declaration is expected ([CTM], Characteristics Declarations).
   EXAMPLE:
   The following query expression will return a TM fragment with persons from the context map which are

younger than `methusalem`. All of these persons get the `homepage` characteristics copied, the name(s) is (are) embedded into the newly generated `comment` characteristics:

```
for $p in // person
where
   is-younger ($p, methusalem / age)
return """

   { $p = }
   { $p / homepage }
   comment: the old name was : { $p / name }

"""
```

If a characteristic is *not* scheduled for atomification, it will remain a characteristic.

EXAMPLE:
The following query expression will return the characteristics themselves, not the URL values within them.

```
select $p characteristics :>: homepage
where
   $p isa person
```

4.6 Tuples and Tuple Sequences

4.6.1 Tuples

Tuples are ordered collections of simple values whereby the values may be of different type (heterogeneous). The types of the individual tuple values can be organized also into a tuple, the *tuple profile*.

A tuple without a single value is called an *empty tuple*. Tuples with only a single value are called *singletons*. Any simple value can be interpreted as singleton and vice versa.

The length of the tuple is called the *arity of the tuple*. As individual values of a tuple are ordered, the first value is assigned the index `0`, the next `1`, etc. Indices can be used to extract the corresponding value from a given tuple. A pair of indices can also be used to extract a slice of values from a tuple. If an index larger or equal the arity of a tuple is used, the extraction will result in the value `null`.

EXAMPLE:
The following query expression assigns iteratively `name`/`homepage` pairs to a variable `@a`. That is then used together with indices to select the type of the name::

```
for @a in // person ( . / name, . / homepage )
return
    (@a[0] *)
```

NOTE:
It is not possible to denote individual tuples, only *tuple sequences* (4.6.3). This implies that indexing and slicing only works with variables (5.1).

4.6.2 Comparing Tuples

The empty tuple is equivalent with the constant `null`. Otherwise, tuples are only then equivalent, if they have the same length and all their individual values are equivalent.

When tuples are to be compared with each other for inequality, this is always done in the context of an *ordering tuple*. Such a tuple has the length of the longer tuple and only has components with values `asc` or `desc`. If that ordering tuple is not explicit, a tuple containing only `asc` values 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 `asc` as value on a given index, that tuple with the smaller component on that index is also the smaller tuple. If the ordering tuple has `desc` 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.

EXAMPLE:
The tuple `(4, "ABC", 3.14)` is smaller than `(4, "DEF", 2.78)`.

EXAMPLE:
The tuple `(4, "ABC", 3.14)` is smaller than `(4, "ABC", 2.78)` under the ordering tuple `(asc, asc, desc)`.

4.6.3 Tuple Expressions

Tuple sequences are sequences of tuples where all tuples have identical profiles. Tuple sequences can be generated with tuple expressions:

[27] *tuple-expression*	→	*tuple-expression* **++** *tuple-expression*	
		tuple-expression **--** *tuple-expression*	
		tuple-expression **=** *tuple-expression*	
		if *tuple-expression* **then** *tuple-expression* [**else** *tuple-expression*]	
		< *value-expression* [*order-direction*] >	
[28] *order-direction*	→	**asc** | **desc**	

Tuple expressions can be combined with the binary operators `=`, `++` and `--`, in the order of precedence. Tuple expressions can also be conditional, in that the *condition tuple expression* (that between `if` and `then`) determines whether final tuple sequence originates from that in the THEN clause (that following `then`), or alternatively, whether the tuple sequence originates from—optional—ELSE clause. If no ELSE clause is provided, `else null` will be used.

EXAMPLE:
The tuple expression `(1, // person)` will return a tuple sequence with the first component the constant value `1` and the second component being a topic item of class `person`. For each person such a tuple exists, but there is no ordering in the sequence.

EXAMPLE:
The tuple sequence `(// person, // person)` contains any 2-combination of topic items of class `person` in the current context map.

EXAMPLE:
The following query expression returns a tuple sequence with one tuple for every person name. The first value is that name, the second is the string `voter` or `non-voter`, depending on the age of this person.

```
for $p in // person
return (
        $p / name,
        ( if $p / age >= 18 then
            "voter"
          else
            "non-voter"
         )
)
```

1. If tuple expressions are combined with a binary operator, first both tuple expression operands are evaluated in the current context. This results in two tuple sequences.
   1. If the operator is `=` then the resulting tuple sequence consists of exactly those tuples which exist in both operand tuple sequences.

      EXAMPLE:
      To filter from a map bound to `%m` all items which are instances of the concept `person`, one can write:

      ```
      %map [ . classes :>: = person ]
      ```

      The map, a tuple sequence, will be iterated through. Inside the filter the first component of every individual tuple is addressed via `.` (which shortcuts `@_[0]`). For this the list of classes (immediate and transitive ones) is computed. That tuple sequence is then compared with a trivial one containing only one singleton tuple with the value `person`.

      EXAMPLE:
      The following query expression lists all the persons names who have the same age as `methusalem`. Note that—even if there were several `age` characteristics—the condition would be satisfied if there were only a single match (exists semantics):

      ```
      select $p / name
      where
         $p / age = methusalem / age
      ```

   2. If the operator is `++` then the resulting tuple sequence consists of all tuples from both operand tuple sequences.
   3. If the operator is `--` then the resulting tuple sequence consists of exactly those tuples which exist in the left operand tuple sequence, but not in the right.

      EXAMPLE:
      To following tuple expression can be used to verify the world is back-and-white:

      ```
      // goodguys = (// person - // evil-evildoer)
      ```

2. When a tuple expression is conditional (if-then-else), then the condition tuple expression is evaluated in the current context. Should there be a single tuple in the resulting tuple sequence, then the tuple expression in the THEN clause is evaluated in the current context and used as a result. Otherwise the tuple expression in the ELSE clause is evaluated.
3. When a tuple sequence is computed from value expressions, all these value expressions are evaluated first in the current context (in no particular order). All these partial results will be interpreted as tuple sequences, whereby simple content will be interpreted as the only component of a singleton. The intermediary result is then a tuple of tuple sequences of tuples of simple content. This structure will be *flattened out* by building the cartesian product. The final result sequence will only contain tuples with simple values.

4.6.4 Ordering Tuple Sequences

Tuple sequences are unordered, unless they are explicitly ordered. *Ordering* of tuples within a sequence implies that there is a partial ordering *occurs-before* defined on tuples. Such ordering may be derived from following sources:

1. If the tuple sequence is generated from a tuple expression in which an *order direction* (`asc` or `desc`) for a component is used, then that sequence will be ordered. For this purpose, components which do not have an order direction will be assumed to have `asc`. Then the ordering defined in 4.6.2 is used.

   EXAMPLE:
   The following query expression selects all `person` instances; for each of these all combinations of `name` and `age` characteristics are generated.

   ```
   select // person ( . / name , . / age desc )
   ```

   Since the second component carries an order direction, the first component's one will default to `asc`. The resulting tuple sequence will then be sorted, first according to the name; when there is a draw, then according to descending age information.

   EXAMPLE:
   The tuple sequence specified by `(// person / birthdate desc)` contains tuples with only a single component. That component contains all birthdates occurrences of all instances of class `person` in the current context map. All these birth dates are sorted in descending order.

2. If the tuple sequence is generated from two tuple expressions, *TS1* and *TS2*, via the binary operators `++` or `--` using an *ordered* context sequence (5.5), then any tuple from *TS1* must occur before any tuple from *TS2*. Any other existing ordering within *TS1* or *TS2* must be honored.

   EXAMPLE:
   The following query expression will return a list of `person` names.

   ```
   for $p in // person
   order by $p / age desc
   return
      ( $p / name )
   ```

   That list is partially sorted, namely according to the person's age. If a person has several names, then these appear in no particular order.

   EXAMPLE:
   The following query expression also selects a list of names, like the query above. This time, though, the *partial* lists of names for a single person is sorted by the name.

   ```
   select $p / name asc
   order by $p / age desc
   where
       $p isa person
   ```

4.6.5 Stringifying Tuple Sequences

Stringification is the process of determining the string representation of a tuple or tuple sequence.

When a tuple is *stringified* its components are first converted into their textual representations. All these representations are then concatenated in the order of their index.

When a tuple sequence is *stringified* then the string representations of the individual tuples will be concatenated. If the tuple sequence is ordered, this order is also carried over.

4.7 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 query expressions (Clause 6) to generate output. These expressions must be properly nested using a balanced pair of brackets `{` and `}`. If the characters `{` and `}` are used as-is, they have to be escaped as `\{` and `\}`.

Query expressions 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 query expressions (including the `{}` bracket pair) are replaced with the empty string, then the XML content must be well-formed.

Ed. Note:
This is not really XML, but maybe close enough? Add <?xml?> and PIs?

[29] *xml-content*	→	*xml-element* | *xml-string*
[30] *xml-element*	→	**<** *gname* { *xml-attribute* } (**/>** | **>** { *xml-content* } **</** *gname* **>**)
[31] *xml-attribute*	→	*gname* **=** *string*
[32] *xml-string*	→	{ [^{<] } [**{** *query-expression* **}** *xml-string*]

EXAMPLE:
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 query expressions (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 `&`, `&quot;`, `<`, `>`, `&apos;`. String content is not delimited by quotes.
- XML content is used as-is.
- Tuple sequences are converted into their string representation (4.6.5).
- TM content is serialized using XTM.

EXAMPLE:
Given the following code,

```
for $s in "Portishead"
for $x in <some>XML code</some>
for $m in <message>{$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>`.

4.8 Topic Map Content

TM content is constructed using CTM [CTM] as basic syntax:

[33]	*tm-content*	→	*ctm-instance*

Additionally to the syntactic rules provided by CTM, query expressions can be embedded by wrapping them into a `{}` bracket pair. This is only allowed at the following positions in the CTM text stream:

1. Wherever a topic or association declaration is expected. The result of the query expression must be a tuple sequence consisting of singelton tuples. In these singletons, every topic item will be injected into the CTM instance. Every association item will be injected into the CTM instance. Every string with identifier syntax will be interpreted as source locator and a topic with such source identifier will be injected into the CTM instance. Every string which follows the syntax of a URI will be interpreted as subject locator; also here a topic will be injected into the CTM instance, using this subject locator.
2. Whereever a topic characteristic is expected. The result of the query expression must be a tuple sequence consisting of singelton characteristic items. All these will be attached to the current topic in the CTM stream.
3. Whereever a string (or string fragment) is expected. The result of the query expression will be converted into its string representation. That result is embedded into the string.
4. Whereever a topic identifier is expected. If the result of the query expression is a singleton containing a topic item, then the source locator for that item is used.

EXAMPLE:
The following query expression iterates over all persons in a map. For each item of class `person` a topic in CTM will be declared. The source locator of the person item will also become the source locator in the generated map. The name characteristics are copied as well. A new occurrence of type homepage is added with the generic URL parameterized by the source locator of the topic.

Additionally, for every person, an association is added which records the fact that that person is employed:

```
for $p in // person
return """

   {$p =}
   {$p / name}
   homepage: http://company.org/{$p}.html

   (is-employed-at)
   employer: bigcorp
   employee: {$p}

"""
```

4.9 Value Expressions

Value expressions are expressions which produce a value:

[34] *value-expression*	→	*value-expression* *infix-operator* *value-expression*	
		prefix-operator *value-expression*	
		function-invocation	
		path-expression	
[35] *infix-operator*	→	**...any defined in the predefined environment...**	
[36] *prefix-operator*	→	**...any defined in the predefined environment...**	

Value expressions are either path expressions, function invocations, or a combination of other value expressions using binary infix or unary prefix operators. The accepted operators, their symbols and their precedence are defined in Clause 7. Since all operators are mapped into their function equivalent, the evaluation of a value expression in the current context is either the evaluation of a path expression or that of a function application.

Additional Notation: The following shorthand allows to test for the existence of values and to use a default value otherwise:

[L] *value-expression*	→	*value-expression-1* **||** *value-expression-2*	
	:==	**(** **if** *value-expression-1* **then** *value-expression-1* **else** *value-expression-2* **)**	

EXAMPLE:
The following expression selects all `person` names. If a person does not have a name

```
select $p / name || "undefined"
where
   $p isa person
```

If the value expression is a path expression, the tuple sequence result of the latter will be the overall result (6.6).

A value expression otherwise is a function invocation:

[38] *function-invocation*	→	*item-reference* **(** *function-arguments* **)**	
[39] *function-arguments*	→	*tuple-expression* | < *identifier* **:** *value-expression* >	

A function is simply addressed via an item reference which will attempted to be resolved in the environment map. It is an error if no such function exists there. The arguments can be either assigned *positional* or *via names*, depending whether the function has been defined with or without a parameter profile (6.3.3).

1. For *positional parameter association* first the tuple expression is evaluated in the current context. It is an error if the result is a tuple sequence with a length other than 1. The individual values of that tuple are then assigned to the formal parameters in the function. The evaluation result of the function becomes the overall result.

   EXAMPLE:
   A function `math:sqrt` would be invoked like this:

   ```
   math:sqrt ($p / age)
   ```

2. For *named parameter association* first all value expressions are evaluated in the current context. It is an error if any of them results in a tuple sequence which does not contain exactly one singleton tuple. Actual and formal parameters are then associated via their name. While it is possible that a formal parameter in the function does not get associated a value, it is an error if an actual parameter does not have a corresponding formal parameter. The evaluation result of the function becomes the overall result.

   EXAMPLE:
   The function defined in 6.3.3 would be called as:

   ```
   nr-accounts (owner: "James Bond", map : %_)
   ```

Functions are completely side-effect free. All information is passed into a function via parameters and only information explicitely returned by the function can be experienced by the caller.

4.10 Boolean Expressions

4.10.1 Structure

WHERE clauses and filters make use of boolean expressions:

[40] *boolean-expression*	→	*boolean-expression* **|** *boolean-expression*	
		boolean-expression **&** *boolean-expression*	
		not *boolean-expression*	
		(*boolean-expression* **)**	
		boolean-primitive	

[41] *boolean-primitive*	→	*boolean*	|
		exists-clause	|
		forall-clause	

Apart from the usual boolean constants and operators, boolean expressions can be nested. FORALL clauses test whether a certain condition—again a boolean expression— is satisfied for all members of a particular sequence of values. An EXISTS clause, in contrast, tests whether a condition is satisfied by at least one of the sequence members. In the case that a predicate has been already defined, its truth value can also be tested.

Before a boolean expression is evaluated all occurrences of the variable `$_` are implicitly *existentially quantified*, i.e. each such occurrence iterates independently over every item in the context map. All combinations of such bindings will be collected into a binding set which will be added to the current context. The results of the evaluation of the boolean expression is then `True` if there is only a single such binding for which the expression is `True`. Otherwise the overall result is `False`.

NOTE:
If other free variables exist in the boolean expression and these variables are not bound to a value in the current context, then the evaluation will result in an error. Such variables are **not** implicitly existentially quantified. This is meant as a safeguard to avoid that accidentially potentially large maps are traversed.

Boolean constants immediately evalute to their value. The operators `&`, `|` and `not` have the usual meaning (non-short-circuit, symmetric), the brackets `()` can be used to override the usual precedence, namely that `&` binds stronger than `|` and that `not` binds stronger than `&`.

EXAMPLE:
As usual, the operators `&` and `|` bind the *immediate* boolean expressions. In the example

```
every $opera in // opera satisfies
  composed-by ($person: composer, $opera: opera) &
  some ...
```

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

```
every $opera in // opera satisfies
  composed-by ($person: composer, $opera: opera) &
some ...
```

4.10.2 EXISTS Clauses

An EXISTS clause allows to test whether a particular condition can be satisfied by at least one value out of a set of values:

[42] *exists-clause*	→	**some** < *variable-association* > **satisfy** *boolean-expression*

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

The EXISTS clause evaluates exactly then to `True` if there exists at least one set of variable bindings for exactly those newly introduced variables for which the evaluation of the boolean expression returns `True`. If there is no such binding set (or the set is empty), the overall result is `False`.

EXAMPLE:
The following boolean expression tests whether in the current context map an opera exists with a libretto written by Pink Floyd.

```
some $opera in // opera satisfy
     $opera <- play -> libretto <- opus -> author = pink-floyd
```

Additional Notation: If the boolean condition itself is not relevant, the following abridged form can be used:

[M] *boolean-expression*	→	**exists** *tuple-expression*
	:==	**some** *@* **in** *tuple-expression* **satisfy** **true**

Additional Notation: This can be further abridged by writing the path expression only:

[N] *boolean-expression*	→	*tuple-expression*
	:==	**exists** *tuple-expression*

EXAMPLE:
The following boolean expression tests whether a given author (captured in `$author` in the current context) is an author, i.e. is involved in an association via the role `author`:

```
exists $author <- author
```

4.10.3 FORALL Clauses

A FORALL clause can be used to test whether all values from a given sequence of values satisfy a certain criterion:

[45] *forall-clause*	→	**every** < *variable-association* > **satisfies** *boolean-expression*

Like for EXISTS clauses, not only a single sequence of values can be specified and can be iterated over for testing; whole sets of bindings can be generated by providing several variables.

A FORALL clause is evaluated in the current context. It evaluates to `True` if for all variable binding sets generated with the variable associations the boolean expression evaluates to `True`. Otherwise it will evaluate to `False`. It also evaluates to `True` if not a single binding set is generated.

EXAMPLE:
The following expression tests whether all politicians are honorable persons.

```
every $p in // politican satisfy
      $p <- person -> trait = honorable
```

If our universe would contain not a single politician, then this boolean expression would evaluated to `True`. Otherwise, only when each of them has (at least one) trait `honorable`, only then the expression is true.

EXAMPLE:
The following boolean expression encodes the statement *everybody loves everyone*:

```
every $p  in // person,
      $p' in // person satisfy
      loves (lover: $p, loved: $p')
```

Note, that it is not necessary that persons love themselves to make above expression true.

NOTE:
Obviously, there is a semantic relationship between EXISTS and FORALL clauses:

[O] *boolean-expression*	→	**every** < *variable-association* > **satisfies** *boolean-expression*
	:==	**not some** < *variable-association* > **satisfy not** *boolean-expression*

4.10.4 Virtual Associations

Boolean expressions can be interpreted as anonymous predicates. The free variables in a boolean expression are then the parameters of the predicate. Depending on the variable bindings in the context, that anonymous predicate either returns `True` or `False`. In other words, a predicate answers the question whether particular values are in a particular relationship or not, and—in this sense— predicates are generalized associations. They can be seen as generators for *virtual associations*, although as such they lack an association type.

Ed. Note:
From a logician's point of view, predicates (and the language to define them) provide *reasoning capabilities* in that new knowledge (relationships) can be derived from existing knowledge. Many languages to express predicates exist, and it is well known that with different expressiveness the reasoning capabilities of the underlying logical inference systems vary; and with that the computational costs.

This draft of TMQL commits itself to particular inferencing method as it introduces here not only AND and OR, but also quantifiers (SOME and EVERY) which have to range over potentially huge data sources.

But what is with implementations/applications which simply do not need it? Web interfaces, for instance?

5 Query Contexts

5.1 Variables

During evaluation, variables are supposed to capture values. To identify a particular variable in a particular lexical scope of a query expression, the variable identifier is prefixed by a *sigil* and postfixed by any number of primes (`'`):

[47] *variable*	→	**/[$@%]\w+'*/** [**[** *integer* **]**]

The sigil (either a `$`, `@` or `%`) signals whether the variable can be bound to either a *simple value* (scalar), a tuple or a tuple sequence. This is directly followed by the variable name, consisting of alphanumeric characters (including `_`). Any number of trailing primes may be attached.

EXAMPLE:
Valid variables are `$a`, `$a'`, `$_`, `@a_long_list_name`. Examples for invalid variables are `x` (sigil missing), `$a-string-world` (dashes not allowed in names) or `@list '` (no blanks before the prime are allowed).

Only for tuple variables a subscripting index is allowed.

EXAMPLE:
Valid variable subscripts are `@a_long_list_name[2]` and `@_[0]`. Invalid subscripts are `$a[2]` and `%a[0]`. The latter will be interpreted as filter postfix (6.6.2).

Additional Notation: As it appears quite frequently that the first (and often only) component of a tuple is to be addressed, we introduce an abbreviation. The shorthand `.` means the same as `@_[0]` and is handy if a tuple has only a single component:

[P]	*variable*	→	**.**
		:==	**@_ [0]**

5.2 Variable Bindings

New variable bindings can be created as part of the query evaluation process. For this purpose a new instance of a variable will be associated with a value computed from a path expression:

[49]	*variable-association*	→	*variable* **in** *path-expression*

In any case, the evaluation of the path expression will result in a tuple sequence. How many variable bindings are produced by such an variable association then depends on the variable sigil.

1. If the sigil is `$`, so that the variable can only hold simple values, then a sequence of variable bindings is generated whereby in each of these the variable is bound to exactly one value of all tuples within the tuple sequence. If the tuple sequence is ordered, then the bindings will be ordered as well accordingly; also every tuple will be iterated following increasing indices.
2. If the sigil is `@`, so that the variable can only hold tuples, then—one by one—the tuples within the tuple sequence are assigned, building individual bindings. If the tuple sequence is ordered the bindings will follow this order.
3. If the sigil is `%`, so that the variable can hold a complete tuple sequences, then this tuple sequence will be assigned; only one binding will be generated.

EXAMPLE:
Creating as many bindings as there are `person` topic in the current context map. Every binding contains `$p` bound to one `person` item.

```
$p in // person
```

EXAMPLE:
Creating a list of all `person` items. The function `fn:concat` takes a tuple sequence and produces another one which contains only a single tuple. That consists of the concatenation of all the original tuples.

```
@p in fn:concat (// person)
```

EXAMPLE:
Computing a new map during evaluations:

```
%newmap in %oldmap + http://example.org/my.onto
```

Once a variable is bound to a particular value in a particular binding, this binding also cannot be changed (immutability of variables). Only new bindings can be generated.

A set of variable bindings is called a *binding set*.

5.3 Variable Semantics

At any point in time the processor keeps a stack of binding sets, the *context*, following the nested evaluation structure of query expressions. Whenever a subquery evaluation starts, a new, initially empty, binding set is pushed onto this context. After the evaluation of this subquery the binding set is popped from the context.

> The value of a particular variable in a given context is determined by a binding for that variable in a binding set which has been added latest to the context. It is an error if a variable is evaluated in a context where no binding exists.

1. Any two occurrences of one and the same variable within a particular query expression must have identical values bound to them in that context.
2. Any two occurrences of two different variables may refer to different or the same values. They are regarded to be independent.
3. If the variable names differ only in the number of primes in their postfix, then their values MUST differ.

EXAMPLE:
`$a`, `$a'` and `$a''` within the same scope can never have the same value assigned. So to find three (different) neighbors, the following will work.

```
...
where
    is-neighbor-of (xtm:subject : $a,   xtm:subject : $a')  &
    is-neighbor-of (xtm:subject : $a',  xtm:subject : $a'') &
    is-neighbor-of (xtm:subject : $a'', xtm:subject : $a)
```

If duplicates are allowed in the result, then choosing completely different variables makes them independent:

```
...
where
    is-neighbor-of (xtm:subject : $a, xtm:subject : $b)
```

5.4 Special Variables

Following special variables are assigned automatically during an evaluation. They cannot be used in variable associations.

`%__` (current meta map)
: The meta map (see 6.3) is initially adopted from the environment. At any nested query expression, this map can be enriched.

`%_` (current context map)
: Whenever a map has been referred to (like in a FROM clause within a SELECT query expression, via a path expression, or via a reification naviation step), it becomes the current contextual map for the currently evaluated query expression.

  All item identifiers and navigation steps are interpreted relative to this map.

`@_` (current tuple)
: Whenever within a path expression a tuple sequence is iterated over in projections or filters, the tuples in the sequence—one by one—become the *current tuple*.

`@#` (current scope)
: The current scope is set by the environment. It is `null` if not defined.

`$#` (current position)
: Whenever a tuple sequence is iterated over explicitly, this variable contains the current position of the tuple in the sequence (counting from `0`). Its value is always the same as the result of the function `fn:position()`.

5.5 Context Ordering

Sequences of contexts can be either ordered or unordered. For unordered sequences the actual order within the sequence bears no relevance; implementations do not have to meet any ordering guarantees.

> Whole contexts can be ordered according to an *ordering tuple expression*. For this purpose, the ordering tuple expression will be evaluated separately for every context. If the resulting tuple expression does not contain a single tuple, the value `null` is used. Also `null` is used if a sequence contains more than one tuple.

> A context *C1* is said to *occur before* another, *C2*, if the *C1*'s single tuple is *smaller* than that of *C2* according to 4.6.2.

5.6 Environment

A TMQL processor is invoked by an—otherwise undefined—*environment*. It remains unspecified how the query expression itself is passed to the processor. Before a query expression is evaluated, though, the environment can hand in additional parametric information to the processor (import).

At this outermost level of a query expression, the environment will provide a binding set to import information into the query process. How this is managed on the implementation level is outside the scope of this International Standard. Any number of variable associations can be provided whereby the following variables have special meanings inside a query expression:

`%map` (default: `null`)
: If a single map is to be queried, it can be assigned to this variable. Conceptually it represents a tuple sequence (4.6.3) consisting of all items in the map. At the start of the query evaluation this map—if defined— will be assigned to the contextual map `%_`.

`@scope` (default: `null`)
: If a query expression is supposed to be evaluated by honoring a particular scope, the scoping topics can be listed in this variable. This tuple will be bound to the variable `@#`.

  Ed. Note:
  This is experimental and has to be used to select associations and characteristics in navigation steps.

`%environment` (default: see 7.2)
: This map contains all ontological background information defined by the TMQL processor itself. Environments are free to add more information as detailed in 6.3.

No other information is imported from the environment.

During a query expression evaluation errorenous situations may arise, at the detection of which, the processor must terminate processing. It remains unspecified how processors signal this to the environment. The list in 7.2 contains a minimal list of such conditions; processors are free to provide more detail and/or other conditions.

Query evaluations may return results into the environment. Again, this International Standard does not constrain how this is achieved and how applications can access the results. It also does not specify whether this happens at the end of an evaluation or during the evaluation itself (such as with lazy evaluation).

6 Query Expressions

6.1 Structure

Every query expression is evaluated in the current context. For the outermost query expression this context is defined by the binding set passed in from the environment. For nested query expressions the context is that of the directly enclosed query expression. Every evaluation of a query expression adds a new binding set, which will be discarded afterwards.

A query can take one of three forms: a SELECT expression (6.4), a FLWR expression (6.5), or a path expression (6.6):

[50] *query-expression*	→	[*ontology-clause*] (*select-expression* | *flwr-expression* | *path-expression*)

The *ontology clause* allows to declare and import additional knowledge into the querying process (6.3).

In terms of expressiveness of search patterns, all styles are effectively equivalent. SELECT and FLWR expressions both make use of path expressions as a sub-language. As subexpressions can contain subqueries. It is also possible to mix the different styles within one larger expression.

> First, the ontology clause will be evaluated in the current context. The resulting map will be merged with the current meta map and will be bound to a new instance of the variable `%_`. This binding will be locally added onto the current context.

> The rest of the query expression is then evaluated in this context. After the result has been computed, all local bindings will be removed from the context.

6.2 Comments

Comments in query expressions are allowed where whitespace characters are allowed (3.1). They are introduced with the character `#` when it appears outside a string or outside a URI. Comments reach until the end of the current line, or until the end of the text stream, whatever comes first.

Comments are completely ignored by the query processor.

6.3 Ontology Clause

6.3.1 Structure

The context map contains all factual knowledge a TMQL processor will use first hand. In many cases, though, additional ontological knowledge can be used to enrich the query processors' understanding of the application domain. Such additional knowledge is provided by

- additional factual data, such as topics and associations,
- functional dependencies, i.e. *functions* denoted in a formal language which are capably of computing a result out of a set of parameters, and
- general ontological constraints and rules.

The *ontology clause* contains a map instance denoted in CTM [CTM]:

[51] *ontology-clause*	→	""" *tm-content* """

The map instance may contain any number of topics and associations. Every ontological information so declared becomes a topic in the meta map of the directly enclosing query expressions. As the meta map can be also queried within TMQL, this mechanism enables for introspection.

A TMQL processor will hereby recognize the following classes:

- `tmql:ontology`: For every topic of this class, the source identifier will be locally registered as *prefix*. Inside the query expression these prefixes can be used for subject indication and subject addressing.
- `tmql:function`: For every topic of such a class, the TMQL processor will locally register a function which can be invoked (4.9) inside the query expression.
- `tmql:predicate`: For every topic of such a class, the TMQL processor will locally register a *virtual association* (typed predicate) which can be referred to in path expressions inside the query expression.

6.3.2 Ontologies

Topics of type `tmql:ontology` are interpreted as representatives for an additional ontology. TMQL processors are free to support any available ontology definition technology. It is not specified by this International Standard how implementations determine which technology is to be used.

These new imported ontologies can be referenced directly, can be indicated by a subject identifier, or can be provided inline, as part of the topic. In any case, the source identifiers of the ontology topics become prefixes, to be used in the follow-up query expression.

EXAMPLE:
The ontology below makes it possible to refer to topics with subject identifiers defined in XTM 1.0 using the `xtm` prefix:

```
"""
xtm isa tmql:ontology ~ http://www.topicmaps.org/xtm/1.0/core.xtm
"""
select $p
where
    xtm:class-instance (xtm:class : person, xtm:instance: $p)
```

EXAMPLE:
The following uses explicitly one particular syntax to load a vocabulary:

```
"""
ltm   isa tmql:ontology                 ~ http://www.ontopia.net/ltm/

opera isa tmql:ontology isa ltm:instance ~ file:operas.ltm
"""
```

EXAMPLE:
The following meta map does reference the ontology connected with the AsTMa= notation and uses this notation to denote one other ad-hoc:

```
"""
astma isa tmql:ontology ~http://psi.topicmaps.com/astma/2.0/

opera isa tmql:ontology isa astma:instance
bn: Opera Adhoc Ontology
body: """

  libretto subclasses opus
  bn: Libretto
"""

"""
```

The defined topics `libretto` and `opus` can be accessed as `opera:libretto` and `opera:opus` inside the follow-up query expression.

6.3.3 Functions

Functions are also part of the ontological knowledge about an application domain. They encapsulate a particular dependency between values in the domain and as such abstract away the internal working of an algorithm for the user of the function. Functions can (and should) be used to manage the complexity of a query.

When a function topic is declared, its source locator is visible throughout the follow-up query expression, but only in places where a function invocation (4.9) is allowed.

NOTE:
Functions cannot be overloaded.

TMQL itself can be used to implement a function body. In this case the body of the function must follow the structure of a RETURN clause (6.5). The formal parameters are determined implicitly as all free variables in the RETURN clause, i.e. those variables which are not bound with FOR, SOME or FORALL clauses.

EXAMPLE:
The function `nr-accounts` returns the number of accounts for a given owner.

```
nr-accounts is-a tmql:function
return: fn:length ( %map // accounts [ . -> owner = $owner ] )
```

The free variables are `%map` and `$owner`. If the function `nr-accounts` is to be invoked inside a query expression, then actual parameters for these two variables have to be provided:

```
...
nr-accounts (owner: "James Bond", map : %_)
```

EXAMPLE:
The function `get-accounts` collects all account topics from a map and finds the respective owner. The map itself is handed over as the sole parameter `%map`. The result is handed back as an XML fragment to the caller.

```
get-accounts is-a tmql:function
description: produce an XML fragment with all accounts
return: """
  <accounts>{
  for $a in %map // account return
      <account>{ $a / owner }</account>
  }</accounts>
"""
```

Implementations are free to support additional implementation languages whereby this International Standard does not mandate how the technology is indicated.

EXAMPLE:
The following function uses Python to determine the string representation of the current date.

```
ctime isa tmql:function
```

```
   return @ python: """
   from datetime import ctime
   return ctime()
   """
```

Ed. Note:
Maybe allow parameter profile to be defined?

6.3.4 Predicates

According to 4.10.4 boolean expressions can be interpreted as typeless virtual associations. Virtual associations are *predicates* in the sense that they express constraints on values in the application domain or to introduce new relationships between values, based on existing associations and other predicates (*inferencing*). For the purpose of generalization, variables are used. When now a given predicate is evaluated in a particular context, the result is either `True` or `False`.

Boolean expressions can be complemented with a name to form a *named predicate*. That name—in Topic Map terms—corresponds to a newly introduced association type. All variable names (without the sigil) become the role types of the virtual association implied by the predicate. A TMQL processor recognizes topics of type `tmql:predicate` within the environmental map.

EXAMPLE:
The following CTM declaration defines a predicate with a new association type `is-a-weaner`:

```
   is-a-weaner isa tmql:predicate
   description: checks whether a person is from Vienna
   where: """
      $person isa person &
      lives-in (city: geo:vienna, person: $person)
   """
```

The only free variable in the boolean expression is `$person` which implies that there is a (new) role type `person`.

Inside a query expression that predicate can be referred to as follows:

```
   ...  select $p where is-a-weaner (person: $p)
```

EXAMPLE:
The following predicate *computes* transitive geographical containment.

```
   is-located-in is-a tmql:predicate
   description: a thing is located either directly or indirectly
   where: """
        geo:is-in (xtm:subject: $thing, geo:location: $location)
      | some $l in // geo:location satisfy
          is-in         (xtm:subject : $thing, geo:location : $l) &
          is-located-in (xtm:subject : $l,     geo:location : $location)
   """
```

The free variables are `$thing` and `$location`, so that the corresponding virtual association has two role types: `thing` and `location`. Consequently, inside a query expressions this predicate can be used as follows:

```
   ...
   is-located-in (xtm:subject: $something, location: paris)
   ...
```

The predicate can also be used in a path expression:

```
   ...
   %_ // city [ . <- location -> xtm:subject = eiffel-tower ]
   ...
```

Here the city 'Paris' will also be part of the result, even when there is no direct fact, that the Eiffel tower is in Paris.

When such a topic is declared—at least conceptually—all associations which can be generated with the following query expression are merged into the current context map.

```
for $var0 in %_,
    $var1 in %_,
    ...
return """
    (predicate-name)
    var0 : $var0
    var1 : $var1
    ...
"""
```

Hereby *predicate-name* is the name of the predicate and *var0*, *var1*, ... are the free variables in the boolean expression of the predicate.

6.4 SELECT Expressions

Query expressions can take the form of SELECT expressions:

[52] *select-expression* → **select** *tuple-expression*
[**from** *value-expression*]
[**where** *boolean-expression*]
[**order by** *tuple-expression*]
[**unique**]
[**offset** *value-expression*]
[**limit** *value-expression*]

If the FROM clause is missing, then `from %_` is assumed, i.e. the current context map will be queried. If the WHERE clause is missing, then `where true` is assumed. If the OFFSET clause is missing `offset 0` is assumed.

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

```
select $composer
   where composed-by (composer : $composer, work : $opera) &
         $opera isa opera
```

EXAMPLE:
The query below finds all combinations of operas and their composer(s) whereby this list of pairs is sorted according to the premiere date of the opera. Later operas appear first. Only the first 10 such pairs are returned.

```
select $opera / name, $composer / name
   where composed-by (composer : $composer, work : $opera) &
         $opera isa opera
   order by $opera / premiere-date desc
   offset 0 limit 10
```

First, the FROM clause is evaluated. As the result is interpreted as map, it must be a tuple sequence of singletons which contain items. This map is bound to a new instance of `%_` and so becomes the context map for this query expression.

Then the value expressions in the optional OFFSET and the LIMIT clauses are evaluated. Their results must be non-negative integers, or `null`. Otherwise an error will be flagged. These values will be bound to the variables `$_lower` and `$_limit`, respectively.

Then all free unbound variables in the WHERE clause (not those in the SELECT clause, and not any occurrence of `$_`) are determined. Each of these variables will be *universally quantified*, i.e. iterate (conceptually) over all items in the context map. For all these variables, all possible binding sets are organized into an unordered sequence whereby the variable semantics (5.3) is honored.

NOTE:
According to this, query expressions which name (yet unbound) variables in the SELECT clause and do not constrain them in a WHERE clause are errorenous. This is meant as a safeguard to avoid that queries accidentially traverse—potentially huge—topic maps.

EXAMPLE:
This query expression is **invalid** as it mentions a variable `$thing` which is not constrained in any way:

```
select $thing from %map
```

If the intention is to get everything from a map, then this has to be made more explicit:

```
select $thing from %map
      where $thing isa xtm:subject
```

One by one, every such binding set will be added to the current context. If no ORDER clause exists, then this sequence of contexts will remain unordered. Otherwise it will be sorted according to 5.5 using the tuple expression in the clause.

EXAMPLE:
In the following query expression the result will be all names of all instance of class `person`:

```
select $p / name
   where $p isa person
   order by $p / age
```

The individual persons are sorted by their age, given that they have such property. Note that the individual names are NOT sorted.

With each context from the sequence of contexts, the boolean expression in the WHERE clause is evaluated (4.10). If the evaluation result is `True`, then that particular context will be used to evaluate the tuple expression within the SELECT clause. If the evaluation is `False` this particular context will be ignored.

As the contexts are tested, their tuple sequences are collected and are concatenated into one tuple sequence. If

the sequence of contexts was unordered, consequently so will be the tuple sequence. Otherwise the concatenationthe tuple sequence will be ordered.

If no UNIQUE clause exists, then the tuple sequence computed above remains unchanged in this step. Otherwise the function `fn:unique` will be applied to it, rendering a new tuple sequence where all duplicate tuples have been eliminated.

Finally, the tuple sequence is subjected to a further function `fn:slice` whereby as parameters the values of `$_lower` and the result of `$_lower + $_limit` are passed in. The result of this function—a vertical slice of the tuple sequence— becomes the overall result of the query expression.

The slicing can be optimized away if the lower index is `0` and the upper is undefined.

6.5 FLWR Expressions

FLWR expressions follow the form of generalized loops. They allow a very high degree of control over which values are used to iterate over and what content is to be generated as result. Due to their syntactic structure, FLWR allow not only to generate tuple sequences to be returned, but also the construction of content in XML and TM form:

[53] *flwr-expression* → { **for** < *variable-association* > }
[**where** *boolean-expression*]
[**order by** < *tuple-expression* >]
return *content*

Only the RETURN clause is obligatory; with it the result content is generated. All the other clauses are optional. If the WHERE clause is missing, the default `where true` is assumed.

A given FLWR expression can contain any number of *FOR clauses*. Several variable associations can be listed in a particular *FOR clause*. This is equivalent to having a dedicated FOR clause for every individual variable. Any introduced variable is visible until the end of this query expression. The order of the individual FOR clauses does not bear any meaning.

EXAMPLE:
The following FLWR expression returns all names of group members:

```
for $p in // person
where
   $p <- member
return
   ( $p / name )
```

EXAMPLE:
The following FLWR expression returns all pairs of (different) person topic items where the persons are members within the same group:

```
for $p in // person
  for $p' in // person   # $p is therefore never the same as $p'
  where
     $p <- member -> member = $p'
  return
     ( $p, $p' )
```

Conceptually, the variable associations inside the FOR clauses are evaluated in lexical order, starting with the first. Every evaluation of a single variable association results in a sequence of variable bindings for the given variable (5.2). One by one, these bindings are added to the current context with which the rest of the variable associations are evaluated. At the end of this process stands a sequence of contexts.

If no ORDER clause exists, then this sequence of contexts will remain unordered. Otherwise it will be sorted according to 5.5.

Each context in the sequence will be subjected to a test provided by the boolean expression in the WHERE clause. If the evaluation renders `True` value, then that context will be kept; all other contexts will be discarded.

The RETURN clause is then used for actually generating content, be it a tuple sequence, a single XML structure, or a TM fragment.

If—due to enclosing FOR clauses—several results have been computed for every iteration, then these partial results are combined into a total result using the operator `+`. For strings this combination is using string concatenation, for tuple sequences, the partial sequences are interleaved, for XML content the individual fragments are combined into a node list and for TM content the fragments are merged.

6.6 Path Expressions

6.6.1 Structure

Path expressions follow a *navigate and filter* paradigm. Starting from given values (atoms or items in a map), navigation steps along defined axes within the context map can arrive at new values. These values then can be filtered according to predicates. All this is organized into postfixes:

[54] *path-expression* → *content* [*postfix chain*] | *association predicate*

The *value designation* results in a tuple sequence of values. *Filters* simply mask out all tuples in the sequence

[55] *postfix-chain* → *postfix* [*postfix-chain*]

which do not satisfy the predicate provided with the filter. The navigational aspect of path expressions is provided by *projections*. Given a particular tuple in the tuple sequence, a projection produces another tuple or—more generally—another tuple sequence.

EXAMPLE:
The following path expression computes a table with two columns. The first contains the name of the author of an document, the other lists the names of the authored documents:

```
%biblio // person ( . / name, . <- author -> document / title )
```

Here we assume that one person has exactly one name and each document also has exactly one title.

The value designation is evaluated in the current context, rendering a tuple sequence. If the postfix chain is empty, then this tuple sequence is also the final result of the path expression. Otherwise, one by one, the postfixes are applied in lexical order. The result of the last postfix application is the result of the path expression.

If the tuple sequence computed with the value designation has been unordered, so will be the resulting sequence. Otherwise, the ordering will be maintained in the sense that all postfix applications are *stable operations*, i.e. the evaluation of every postfix occurs in the order of the incoming tuple sequence.

6.6.2 Filter Postfix

With a *filter postfix* conditions on tuples can be defined:

[57]	*filter*	→	**[** *boolean-expression* **]**

When the filter 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 bound to a new instance of `@_`. This binding is added to the context for the evaluation of the boolean expression only.

Only tuples in the incoming tuple sequence where the evaluation of the boolean expression returns a `True` value will be included of the outgoing tuple sequence.

Additional Notation: To test whether the current item is of a particular type or in a particular scope, the following shorthands are provided:

[Q]	*boolean-primitive*	→	***** *item-reference*
		:==	**. classes :>: =** *item-reference*
[R]	*boolean-primitive*	→	**@** *item-reference*
		:==	**. @ =** *item-reference*

Additional Notation: If the condition only tests for a particular position in the tuple sequence, then the usual slice syntax can be used as well:

[S]	*boolean-primitive*	→	**[** *integer* **]**
		:==	**[$# =** *integer* **]**
[T]	*boolean-primitive*	→	**[** *integer-1* **..** *integer-2* **]**
		:==	**[** *integer-1* **<= $# & $# <** *integer-2* **]**
[U]	*boolean-primitive*	→	**[** *integer-1* **..]**
		:==	**[** *integer-1* **<= $#]**

Additional Notation: To filter items of a particular type from a singleton tuple sequence, one can also use the following:

[V]	*postfix-chain*	→	**//** *item-reference* [*postfix-chain*]
		:==	**[*** *item-reference* **]** [*postfix-chain*]

EXAMPLE:
Filter out all `person` instances from the map bound to `%map`.

```
%map // person
```

Additional Notation: If the map to be queried is the context map, then also the map can be omitted:

[W]	*path-expression*	→	**//** *item-reference* [*postfix-chain*]
		:==	**%_ //** *item-reference* **]** [*postfix-chain*]

6.6.3 Projection Postfix

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

[65]	*projection*	→	**(** *tuple-expression* **)**

This postfix is applied to every individual tuple in the incoming tuple sequence.

For every tuple in the incoming tuple sequence, this tuple will be bound to a new instance of `@_`. This binding is added to the current context. Then the specified projection is evaluated in that context. The result of this process is a tuple sequence. As the evaluation is repeated for all incoming tuples, the overall result is the interleaved combination of all these partial result sequences.

EXAMPLE:
Given a map in `%map`, the path expression `%map // opera ( . / name )` extracts first all `opera`s from `%map`. Then for every topic item a new tuple will be built which contains the sequence of names for that opera. The result is the sequence of all names of all `opera`s in the map.

EXAMPLE:
Given a map in `%map`, the path expression `%map // opera ( . / name [ @opus ], . <- work [ * is-composed-by ] )` extracts first all instances of `opera`s. Then a new tuple is (maybe, see below) generated for every such opera. It contains as a first component only the `opus` number as provided by a name in the respective scope. The second tuple component takes the opera as starting point and finds association item(s) of type `is-composed-by` where that particular opera playes the role `work`.

If there are several `is-composed-by` associations for a particular opera, then for each a separate tuple is created. If there is not a single one, then not a single tuple will be generated for that opera. This, of course, also would happen if that opera had no opus number at all.

6.6.4 Association Predicates

Association templates allow to define path postfixes which look for associations of a certain type and having particular role/player combinations.

NOTE:
While association templates are a more concise notation for their purpose than path expressions, they do not introduce new expressitivity or semantics as they can be transformed into path expressions (see 9.1). Nevertheless we will describe their informal semantics here.

[66]	*association-predicate*	→	*item-reference* **(** < *role* **:** *player* > **[, ...]** **)**
[67]	*role*	→	*item-reference*
[68]	*player*	→	*path-expression*

The `item-reference` in the association predicate is interpreted to be an association type in the context map. Only such associations are considered to be in the results. Any number of role/player combinations can be specified. The roles are all item identifiers, the players are computed via a path expression. The optional ellipsis `...` can be used to indicate that matching associations may contain other role/player not mentioned.

In the current context all path expressions for the players are evaluated. The result of each such evaluation is a tuple sequence. All these sequences must contain singleton tuples with topic items as values; otherwise an error is flagged. For one role this is kept as list of *potential players*.

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

1. The association item must be an instance of the given type in the context map, honoring class transitivity.
2. For each of the roles mentioned in the association template, that item must have a role which is a subclass (directo or indirect) of the role specified; and for that very role it must have a player out of the sequence of potential players.
3. If the ellipsis `...` has not been used, there must not exist further role/players in the association item which are not explicitely named in the association predicate.

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

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

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

The result is a tuple sequence of association items where any of the cities in the map have a theatre.

Additional Notation: Following shorthand notations for *subclass-of* and for *instance-of* can be used:

[X]	*association-predicate*	→	*simple-content-1* **subclasses** *simple-content-2*
		:==	**xtm:subclass-of (xtm:subclass :** *simple-content-1* **, xtm:superclass :** *simple-content-2* **)**
[Y]	*association-predicate*	→	*simple-content-1* **isa** *simple-content-2*
		:==	**xtm:instance-of (xtm:instance :** *simple-content-1* **, xtm:class :** *simple-content-2* **)**

EXAMPLE:

While roles cannot be omitted, the wildcard `*` can be used instead if the role does not matter.

```
composed-by (composer: vivaldi, * : opera)
```

NOTE:
The use of `*` for role typs may have a negative impact on optimizers.

7 Predefined Environment

7.1 Please Read

The following is just a collection of ideas and some experiments.

7.2 TMQL Concepts

This map defines the main concepts of TMQL.

```
# defining the target namespace
tmql      ~ http://www.isotopicmaps.org/tmql/1.0 ~ ctm:self
bn: TMQL
bn @long: Topic Maps Query Language
in: textual language to extract content from TM-based backends

#-- core concepts ------------------------------------------------------

ontology isa tmql-concept
bn: Ontology
in: """
A TMQL ontology is a topic map. Any topic of this type is automatically
recognized by a TMQL processor, in that the topic identifier is registered
as prefix. The subject identifier(s) correspond to the namespace URI associated
with the prefix (6.3).
"""

function isa tmql-concept
bn: Function
in: computes a new value from existing ones
in: there are predefined functions and user-defined ones

predicate isa tmql-concept
bn: Predicate
in: stands for a collection of associations
in: parameterized

#-- Errors -------------------------------------------------------------

tmql-processing-error isa tmql-concept
bn: TMQL Processing Error
in: an errorenous situation which has to be flagged to the application

unidentified-subject         isa tmql-processing-error

unbound-variable-dereference isa tmql-processing-error

uri-dereferencing-problem    isa tmql-processing-error

type-coercion-problem        isa tmql-processing-error

#-- data typing --------------------------------------------------------

datatype
bn: Data Type

primitive-datatype subclasses datatype
bn: Primitive Data Type

boolean isa datatype ~ http://www.w3.org/TR/xmlschema-2/datatypes.html#boolean
bn: Boolean
in: value is either 'True' or 'False'

integer isa datatype ~ http://www.w3.org/TR/xmlschema-2/datatypes.html#integer
bn: Integer Number
in: positive, negative and 0

decimal isa datatype ~ http://www.w3.org/TR/xmlschema-2/datatypes.html#decimal
bn: Decimal Numbers
in: with a fractional part

date    isa datatype ~ http://www.w3.org/TR/xmlschema-2/datatypes.html#dateTime

uri     isa datatype ~ http://www.w3.org/TR/xmlschema-2/datatypes.html#anyURI

string  isa datatype ~ http://www.w3.org/TR/xmlschema-2/datatypes.html#string
bn: Strings
syntax: """
```

```
            Strings are character sequences terminated by ".




   1.  the string value does not contain the terminating quotes,


   2.  any occurrence of the character " inside the string has to be escaped as
               \", and


   3.  Any occurrence of the character \ itself must be escaped as
               \\


   4.  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.




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


   """^^xsd:html

   complex-datatype subclasses datatype
   bn: Complex Data Type

   xml   isa complex-datatype
   bn: XML Content

   tuple isa complex-datatype
   bn: Tuple Content
   in: ordered collection of primitive values

   tuple-sequence isa complex-datatype
   bn: Tuple Sequence
   in: a sequence of tuples
   in: can be ordered, or not
```

7.3 TMDM Concepts

This map defines some basic vocabulary to deal with TMDM items. It also contains an explicit version of the transitive subclass-of and instance-of relationship, formalized as TMQL predicates.

Ed. Note:
Maybe this can go elsewhere?

```
   tmql isa http://www.isotopicmaps.org/tmql/1.0#ontology ~ http://www.isotopicmaps.org/tmql/1.0

   # target namespace???????

   #== TMDM ========================================================================

   topic                                          ~ http://www.topicmaps.org/xtm/1.0/#psi-topic

   #-- TMDM predefined characteristics ------------------------

   association                                    ~ http://psi.topicmaps.org/sam/1.0/#association

   has-characteristic subclasses association
   bn: has characteristics

   has-occurrence subclasses has-characteristic   ~ http://psi.topicmaps.org/sam/1.0/#occurrence
   bn: has occurrence

   has-basename subclasses has-characteristic     ~ http://psi.topicmaps.org/sam/1.0/#topic-name
   bn: has basename
```

```
has-uri-occurrence subclasses has-occurrence
bn: has URI occurrence

has-data-occurrence subclasses has-occurrence
bn: has data occurrence

#-- TMDM predefined predicates -------------------------------------

is-subclass-of-transitive isa tmql:predicate
bn: transitive (and reflexive) version of is-subclass-of
body: """
  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-transitive isa tmql:predicate
bn: transitive (and reflexive) version of instance-of
body: """
  instance-of ($INSTANCE, $TYPE) |

  instance-of ($INSTANCE, $SUB) &
  is-subclass-of-transitive ($SUB, $TYPE)
"""
```

7.4 Types and Functions

```
# target namespace: http://www.topicmaps.org/tmql/1.0/functions


tmql isa http://www.isotopicmaps.org/tmql/1.0#ontology ~ http://www.isotopicmaps.org/tmql/1.0


#== Functions ==================================================================

type (x) -> datatype



#-- Boolean ------------------------------------------------------------------------

tmql-boolean-and isa tmql:function
bn: ,
in (profile): ($a as BOOLEAN, $b as BOOLEAN) as BOOLEAN

tmql-boolean-or isa tmql:function
bn: ||
in (profile): ($a as BOOLEAN, $b as BOOLEAN) as BOOLEAN

tmql-boolean-not isa tmql:function
bn: not
in (profile): ($a as BOOLEAN) as BOOLEAN


#-- Integer ------------------------------------------------------------------------



           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 >=.



tmql-number-binary-add isa tmql:function
bn: +
in (profile): ($a as NUMBER, $b as NUMBER) as NUMBER

tmql-number-binary-minus isa tmql:function
bn: -
in (profile): ($a as NUMBER, $b as NUMBER) as NUMBER

tmql-number-binary-mul isa tmql:function
bn: *
in (profile): ($a as NUMBER, $b as NUMBER) as NUMBER

tmql-number-binary-div isa tmql:function
bn: /
in (profile): ($a as NUMBER, $b as NUMBER) as NUMBER
```

```
tmql-number-unary-add isa tmql:function
bn: +
in (profile): ($a as NUMBER) as NUMBER

tmql-number-unary-minus isa tmql:function
bn: -
in (profile): ($a as NUMBER) as NUMBER


< <= = > >=
+ - * div mod

-


#-- Decimal ------------------------------------------------------------------------


#-- Date ---------------------------------------------------------------------------

#-- URI ----------------------------------------------------------------------------


#-- String -------------------------------------------------------------------------



tmql-string-concat isa tmql:function
bn: concat
bn @ infix: +
in (profile): ($a as STRING, $b as STRING) as STRING

tmql-string-length isa tmql:function
bn: length
in (profile): length ($s as STRING) as NUMBER

tmql-string-less-than isa tmql:function
bn @ infix : <
in (profile): ($a as STRING, $b as STRING) as BOOLEAN

tmql-string-less-equal isa tmql:function
bn @ infix : <=
in (profile): ($a as STRING, $b as STRING) as BOOLEAN

tmql-string-greater-equal isa tmql:function
bn @ infix : >=
in (profile): ($a as STRING, $b as STRING) as BOOLEAN

tmql-string-greater-than isa tmql:function
bn @ infix : >
in (profile): ($a as STRING, $b as STRING) as BOOLEAN

tmql-string-equal isa tmql:function
bn @ infix : =
in (profile): ($a as STRING, $b as STRING) as BOOLEAN

tmql-string-not-equal isa tmql:function
bn @ infix : !=
in (profile): ($a as STRING, $b as STRING) as BOOLEAN


#-- regular expressions ------------------------------------

tmql-regexp-match isa tmql:function
bn: match
bn @ symbol : =~
in (profile): ($s as STRING, $re as STRING, $sw) as BOOLEAN

tmql-regexp-not-match isa tmql:function
bn: not match
bn @ symbol : !~
in (profile): ($s as STRING, $re as STRING, $sw) as BOOLEAN



#-- XML ----------------------------------------------------------------------------

tmql-xml-add isa tmql:function
bn: +
in (profile): ($a as XML, $b as XML) as XML
in: combines two XML node sequences into one


#-- Tuples and Tuple Sequences -----------------------------------------------------

tmql-interleave isa tmql:function
bn: ||
```

```
in (profile): (%a, %b)

tmql-fold-ts isa tmql:function
in (profile): (%a)

tmql-fold-t isa tmql:function
in (profile): (@a)

unique (TS) -> TS
vfold (TS) -> tuple
hfold (TS) -> TS

# comprehension
vfold (TS, binaryop?)
hfold (TS, binaryop?)

concat (TS) -> tuple


@@@@ add folding

        A tuple sequence can be folded into one tuple. This is done component-wise by using
        the binary operator + to combine all corresponding tuple components.



 If the tuple sequence specified by (1, // person) is to be folded, the first
         component of the resulting tuple would contain the exact number of all person
         instances in the current context map. The second component would contain all items which are
         (directly or indirectly) instances of person.  @@@@ CONFLICT with 'tuples can
         only contain simple things as components.@@@@@




        Tuples can be folded into simple values. For this purpose the binary operator
        + is used to combine all values of the tuple (list comprehension).


#-- Maps ------------------------------------------------------------------------

        TM content can also be built by combining any number of maps using the operator +. It symbolize
        the generic merge operation as defined in [TMDM].

# + * operations for items and whole maps
```

8 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.

9 Formal Semantics

9.1 Mapping Association Predicates to Path Expressions

@@@@@@@@@@ TBW @@@@@@@@@@@@@@@@@@@@@@@@

9.2 Mapping TMDM to TMRM Instances

@@@@@@@@@@ TBW @@@@@@@@@@@@@@@@@@@@@@@@

A Delimiting Symbols (normative)

```
@$% ^ & *  - + (){}[] "'/\ <>, .~|=
```

B Syntax (informative)

Core Syntax

[66]	*association-predicate*	→	*item-reference* **(** < *role* **:** *player* > **[, ...]** **)**
[X]	*association-predicate*	→	*simple-content-1* **subclasses** *simple-content-2*

		:==	**xtm:subclass-of (xtm:subclass : *simple-content-1* , xtm:superclass : *simple-content-2*)**
[Y]	*association-predicate*	→	*simple-content-1* **isa** *simple-content-2*
		:==	**xtm:instance-of (xtm:instance : *simple-content-1* , xtm:class : *simple-content-2*)**
[2]	*atom*	→	**null** | *boolean* | *integer* | *decimal* | *uri* | *date* | *string* [**^^** *qname*]
[16]	*axis*	→	**classes** | **superclasses** | **players** | **roles** | **characteristics** | **scope** | **reifies** | **address** | **indicator** | **atomify**
[3]	*boolean*	→	**True** | **False**
[40]	*boolean-expression*	→	*boolean-expression* **|** *boolean-expression* | *boolean-expression* **&** *boolean-expression* | **not** *boolean-expression* | **(** *boolean-expression* **)** | *boolean-primitive*
[M]	*boolean-expression*	→	**exists** *tuple-expression*
		:==	**some** *@* **in** *tuple-expression* **satisfy true**
[N]	*boolean-expression*	→	*tuple-expression*
		:==	**exists** *tuple-expression*
[O]	*boolean-expression*	→	**every** < *variable-association* > **satisfies** *boolean-expression*
		:==	**not some** < *variable-association* > **satisfy not** *boolean-expression*
[41]	*boolean-primitive*	→	*boolean* | *exists-clause* | *forall-clause*
[Q]	*boolean-primitive*	→	* *item-reference*
		:==	**. classes :>: =** *item-reference*
[R]	*boolean-primitive*	→	**@** *item-reference*
		:==	**. @ =** *item-reference*
[S]	*boolean-primitive*	→	**[** *integer* **]**
		:==	**[$# = ***integer* **]**
[T]	*boolean-primitive*	→	**[** *integer-1* **..** *integer-2* **]**
		:==	**[** *integer-1* **<= $# & $# <** *integer-2* **]**
[U]	*boolean-primitive*	→	**[** *integer-1* **..]**
		:==	**[** *integer-1* **<= $#]**
[0]	*content*	→	*simple-content* | **(** *tuple-expression* **)** | **"""** *tm-content* **"""** | *xml-content* | **{** *query-expression* **}**
[6]	*date*	→	**...xsd:dateTime...**
[5]	*decimal*	→	`[ - ] [0-9]+ [ . [0-9]+ ]`
[42]	*exists-clause*	→	**some** < *variable-association* > **satisfy** *boolean-expression*
[57]	*filter*	→	**[** *boolean-expression* **]**
[53]	*flwr-expression*	→	{ **for** < *variable-association* > } [**where** *boolean-expression*] [**order by** < *tuple-expression* >] **return** *content*
[45]	*forall-clause*	→	**every** < *variable-association* > **satisfies** *boolean-expression*
[39]	*function-arguments*	→	*tuple-expression* | < *identifier* **:** *value-expression* >
[38]	*function-invocation*	→	*item-reference* **(** *function-arguments* **)**
[12]	*identifier*	→	**/\w[\w\-\.]*/**
[35]	*infix-operator*	→	**...any defined in the predefined environment...**

[4]	*integer*	→	`[0-9]+`
[9]	*item-reference*	→	*identifier* | *qname*
[A]	*item-reference*	→	*
		:==	**xtm:subject**
[14]	*navigation*	→	*step* [*navigation*]
[J]	*navigation*	→	**/** *item-reference*
		:==	**characteristics :>:** *item-reference* **atomify :>:** *
[K]	*navigation*	→	**** *item-reference* [*navigation*]
		:==	**atomify :<:** * **characteristics :<:** *item-reference* [*navigation*]
[51]	*ontology-clause*	→	**"""** *tm-content* **"""**
[28]	*order-direction*	→	**asc** | **desc**
[54]	*path-expression*	→	*content* [*postfix-chain*] | *association-predicate*
[W]	*path-expression*	→	**//** *item-reference* [*postfix-chain*]
		:==	**%_ //** *item-reference* **]** [*postfix-chain*]
[68]	*player*	→	*path-expression*
[56]	*postfix*	→	*filter* | *projection*
[55]	*postfix-chain*	→	*postfix* [*postfix-chain*]
[V]	*postfix-chain*	→	**//** *item-reference* [*postfix-chain*]
		:==	**[*** *item-reference* **]** [*postfix-chain*]
[11]	*prefix*	→	**/\w+:/**
[36]	*prefix-operator*	→	**...any defined in the predefined environment...**
[65]	*projection*	→	**(** *tuple-expression* **)**
[10]	*qname*	→	*uri* | *prefix identifier*
[50]	*query-expression*	→	[*ontology-clause*] (*select-expression* | *flwr-expression* | *path-expression*)
[67]	*role*	→	*item-reference*
[52]	*select-expression*	→	**select** *tuple-expression* [**from** *value-expression*] [**where** *boolean-expression*] [**order by** *tuple-expression*] [**unique**] [**offset** *value-expression*] [**limit** *value-expression*]
[1]	*simple-content*	→	(*atom* | *item-reference* | *variable*) [*navigation*]
[15]	*step*	→	*axis* (**:>:** | **:<:**) [*item-reference*]
[B]	*step*	→	**instances :>:** *item-reference*
		:==	**classes :<:** *item-reference*
[C]	*step*	→	**subclasses :>:** *item-reference*
		:==	**superclasses :<:** *item-reference*
[D]	*step*	→	**->** *item-reference*
		:==	**players :>:** *item-reference*
[E]	*step*	→	**<-** *item-reference*
		:==	**players :<:** *item-reference*
[F]	*step*	→	**@**
		:==	**scope :>:** *
[G]	*step*	→	**=**
		:==	**address :<:** *
[H]	*step*	→	**~**
		:==	**indicator :<:** *
[I]	*step*	→	**>>**
		:==	**reifies :>:** *
[8]	*string*	→	**"** { [^"] } **"**
[33]	*tm-content*	→	*ctm-instance*
[27]	*tuple-expression*	→	*tuple-expression* **++** *tuple-expression* | *tuple-expression* **--** *tuple-expression* | *tuple-expression* **=** *tuple-expression* | **if** *tuple-expression* **then** *tuple-expression* [**else** *tuple-expression*] | < *value-expression* [*order-direction*] >
[7]	*uri*	→	**...xsd:anyURI...**

[34]	*value-expression*	→	*value-expression* *infix-operator* *value-expression* | *prefix-operator* *value-expression* | *function-invocation* | *path-expression*
[L]	*value-expression*	→	*value-expression-1* **||** *value-expression-2*
		:==	**(** **if** *value-expression-1* **then** *value-expression-1* **else** *value-expression-2* **)**
[47]	*variable*	→	**/[$@%]\w+'*/** [**[** *integer* **]**]
[P]	*variable*	→	**.**
		:==	**@_** **[** **0** **]**
[49]	*variable-association*	→	*variable* **in** *path-expression*
[31]	*xml-attribute*	→	*qname* **=** *string*
[29]	*xml-content*	→	*xml-element* | *xml-string*
[30]	*xml-element*	→	**<** *qname* { *xml-attribute* } (**/>** | **>** { *xml-content* } **</** *qname* **>**)
[32]	*xml-string*	→	{ [^{<] } [**{** *query-expression* **}** *xml-string*]

Bibliography

TMQLreq, *TMQL Requirements*, ISO, 2003

TMQLuc, *TMQL Use Cases*, ISO, 2003

AsTMa, *AsTMa= 2.0 Specification*, Bond University, 2006, http://astma.it.bond.edu.au/astma=-spec-2.0r1.0.dbk