Topic Maps Query Language

2007-07-13

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 document provides syntax to form valid query expressions and also an informal and a formal semantics for every syntactic form.

To constrain the interaction and information flow between a querying application and a TMQL query processor (short: *processor*) this International Standard also describes an abstract processing environment, loosely defines the passing of parameters into the query process and the exchange of result values. This environment also includes a minimal, predefined set of functions and operators every conformant processor must provide.

This International Standard does provide means for importing external ontologies and additional functionality.

This International Standard does not define an API (application programming interface) to interact with query processors. It 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 5.0.0*, The Unicode Consortium, Reading, Massachusetts, USA, Addison-Wesley Developer's Press, 2007, ISBN 0-321-48091-0, http://www.unicode.org/versions/Unicode5.0.0/

TMDM, *ISO 13250-2 Topic Maps — Data Model*, ISO, 2006, Lars Marius Garshol, Graham Moore, http://www.isotopicmaps.org/sam/sam-model/

TMRM, *ISO 13250-5 Topic Maps — Reference Model*, 2007, Patrick Durusau, Steve Newcomb, Robert Barta, http://www.isotopicmaps.org/tmrm/

CTM, *ISO 13250-6 Topic Maps — Compact Syntax*, Gabriel Hopmans, Lars Heuer, Sam Oh, Steve Pepper, http://www.semagia.com/tmp/ctm.html

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

XTM 2.0, *Topic Maps ? XML Syntax*, ISO 13250: Topic Maps, Lars Marius Garshol, Graham Moore, http://www.isotopicmaps.org/sam/sam-xtm/

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*, T. Berners-Lee, R. Fielding, L. Masinter, 2005, http://www.ietf.org/rfc/rfc3986

RFC3987, *RFC3987 - Internationalized Resource Identifiers (IRIs)*, M. Duerst, M. Suignard, 2005, http://www.ietf.org/rfc/rfc3987.txt

RegExp, *IEEE Std 1003.1, 2004 Edition*, 2004, The Open Group Base Specifications Issue 6, http://www.opengroup.org/onlinepubs/009695399/mindex.html

3 Notation and Semantics

3.1 Syntax Conventions

The syntax is defined on three levels:

1. At the *token level* this International Standard makes use of regular expressions [RegExp] to specify case-sensitive character patterns for valid terminal symbols. They are enclosed in `//` to contrast them to constant tokens. Specifically, it is using `\w` for the character class `[a-zA-Z0-9_]` (alphanumeric characters and the *underscore* character) and `\d` for digit characters `[0-9]`. Tokens are rendered in the text in bold and are underlined.
   Tokens not defined in the grammar are those for binary infix and unary prefix operators; these are specified in the predefined environment together with their function counterparts (Annex A).
   As usual, character symbols are either *delimiting* or not. The list of delimiting symbols is provided in Annex B. 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 tokens; this is not encoded explicitly in the syntax. Whitespace characters are insignificant except within strings and within XML fragments.
2. The *canonical syntax level* is defined using a context-free grammar ([XML 1.0]) with the following conventions: For `A *` we use `{ A }`, for `A ?` we use `[ A ]` and for `( A ( ',' A )* )?` (a comma-separated, possibly empty list) we use `< A >`. Productions for non-terminals are numbered for reference.
3. On top of the canonical syntax, a *non-canonical syntax level* is 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) on the right-hand side of a production is expanded (using term substitution `==>`) into another term. Any abbreviated form is semantically equivalent to its expanded form, so shortcuts do not add any computational complexity to the language but are only added for convenience. These mappings are numbered with letters for reference.

Comments are fragments of the character stream which are ignored by any processor. Comments are allowed where whitespace characters are allowed and are introduced by a hash (#) character (1) at the very beginning of a line or (2) a hash character following a whitespace character outside a string or XML fragment. Comments reach until the end of the current line; or until the end of the text stream, whichever comes first. Comments are not made explicit in the grammar.

Annex C contains the complete language syntax. This 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.

3.2 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 (as is this paragraph); it — by its nature — is ambiguous and is only meant to support the human reader.

The informal semantics is superceded by the formal semantics which is detailed in Clause 7.

3.3 Ontological Commitments

A TMQL processor assumes that the environment which provides access to the queried map(s) will also provide *type transitivity* and *type membership*:

1. For the binary relation between a subtype and a supertype the interpretation and representation in [TMDM] (7.3) is adopted. Accordingly, for *transitivity* a TMQL processor will assume that if a concept B is a supertype of A, and C is a supertype of B, then also C is a supertype of A. A processor will also interprete such relation as *reflexive* so that every type is a subtype and a supertype of itself.
2. For the binary relation between an instance and a type the interpretation and representation in [TMDM] (7.2) is adopted. Accordingly, a processor will assume that if a concept is an instance of a type C, that very concept is also an instance of all supertypes of C.

Apart from this, a TMQL processor is *entailment neutral*, i.e. it leaves it to the environment to perform any inferencing.

This International Standard does not define specific error conditions, be it for errors during parsing TMQL expressions or be it for errors during the evaluation of a query.

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

tm
http://psi.topicmaps.org/iso13250/model/

This is the namespace for the concepts defined by TMDM.

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

This is the namespace for the XML Schema Datatypes.

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

Under this prefix the concepts of TMQL itself are located.

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

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

4 Content

4.1 Constants

Constants are either atomic values from the sets of predefined basic data types, or they can also be references to items in a topic map:

```
[1]   constant          ::=   atom | item-reference
```

4.2 Atoms

Atoms are literal values, such as strings, integer or dates which can be used as constants. This International Standard adopts a list of primitive data types from [CTM] together with operators for these types.

```
[2]   atom              ::=   undefined          |
                              boolean            |
                              integer            |
                              decimal            |
                              iri                |
                              date               |
                              dateTime           |
                              string [ ^^ QIRI ]
[3]   undefined         ::=   undef
[4]   boolean           ::=   true  |  false
[5]   integer           ::=   /[+-]?\d+/
[6]   decimal           ::=   /[+-]?\d+(\.\d+)?/
[7]   date              ::=   ...http://www.w3.org/2001/XMLSchema#date...
[8]   dateTime          ::=   ...http://www.w3.org/2001/XMLSchema#dateTime...
[9]   iri               ::=   <  QIRI  >  | QIRI
[10]  string            ::=   /"([^"]|\")*"/  |  /'([^']|\')*'/
```

Obviously, decimal patterns are preferred over the shorter integer patterns. IRI references may be optionally wrapped by a <> bracket pair (without any white spaces). Strings may use either " or ' as terminators (again without any further white spaces) whereby the terminator can be escaped within the string with a leading backslash (\).

EXAMPLE:
The following are valid atoms (type in brackets):

```
23                                      (integer)
3.1415                                  (decimal)
"Hello World"                           (string)
'Dante\'s Hell'                         (string)
2005-10-16T10:29Z                       (date)
http://example.org/something            (IRI)
<http://example.org/something>          (IRI)
undef                                   (undefined)
```

EXAMPLE:
The following are invalid atoms:

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

For all types, an IRI (or a QName) can be explicitly provided to indicate the data type. This allows implementations to offer additional primitive data types (Clause 8).

EXAMPLE:
The following are also valid atoms (type in brackets):

```
"23"^^xsd:integer                          (xsd:integer)
"http://example.org/something"^^xsd::string  (xsd:string)
"#@**(b($^o$@$nd"^^what:ever               (what:ever)
```

For references, either absolute IRIs or QNames can be used:

[11]	*QIRI*	::=	*IRI* | *QName*
[12]	*IRI*	::=	**([^<>'{}|^`] - [#x00-#x20])***
[13]	*QName*	::=	*prefix identifier*
[14]	*prefix*	::=	**/\w+:/**
[15]	*identifier*	::=	**/\w[\w\-\.]*/**

IRIs are strings not containing angle brackets, curly brackets, the pipe character (`|`) and the caret (`^`) as well as no non-printable character.

For QNames the prefix (without the trailing colon) has to be declared as such by the environment (6.3). No blanks between the prefix and the following identifier are allowed.

NOTE:
Prefixes cannot be empty, so there is no mechanism to use identifiers relative to a document base.

If the item reference is a *QName*, then the QName prefix (without the trailing colon `:`) is interpreted as an item reference for an item of type `tmql:ontology` in the effective map; it is an error, if no such topic exists. If no subject indicator for that ontology topic exists, then an error will be flagged. Otherwise, one such subject indicator — together with the identifier in the QName — is used to construct an absolute IRI according to the rules in [RFC3986] (5.2, Relative Resolution).

4.3 Item References

In the map to be queried (*effective map*, 6.2) Topic Map items can be name directly by an *item identifier*, or a *subject identifier*, as provided by an IRI or a QName:

[16]	*item-reference*	::=	*identifier* | *QIRI*

1. If the item reference is an *identifier* then this identifier is interpreted as an *item identifier* ([TMDM], Clause 5.1) for an item. The result is then this item; if no such item exists, an error will be flagged.
   EXAMPLE:
   The following expression first identifies the topic with the item identifier `jack` and then retrieves all its names:

   ```
   jack / name
   ```

   NOTE:
   While convenient, item references are not a robust way to identify topics. TMDM processors are not constrained how they assign item identifiers to items ([TMDM], Clause 5.1).
2. If the item reference is a QName, then first that is expanded into an absolute IRI according to 4.2. That absolute IRI is interpreted as *subject indicator* for a topic in the effective map. If no such topic exist, an error will be flagged.
   EXAMPLE:
   The prefix in the QName `tm:subject` is `tm`. That is the item identifier for a topic in the predefined environment (Annex A). That topic has a subject identifier `http://psi.topicmaps.org/iso13250/glossary/` which — together with the identifier `subject` will be used to form `http://psi.topicmaps.org/iso13250/glossary/subject`, which is further used as subject indicator in the effective map.
   EXAMPLE:
   The item reference `http://example.org/something/` is interpreted as subject identifier.

Additional Notation: The shorthand `*` always indicates the same as `tm:subject`.

[A]	*item-reference*	::=	*****
		==>	**tm:subject**

EXAMPLE:
`tm:subject` or its short form `*` can be used when the particular topic is not relevant, or unknown. As any Topic Map item represents an instance of *subject* by definition, this makes predicates such as `is-located-in (* : $place, location: paris)` more robust. It may impact performance, though, as no index can be used.

4.4 Navigation

Each navigation step is interpreted within the effective map (6.2). Navigational axes are derived from the structure of a Topic Map instance [TMDM] and can either be followed in *forward* (`>>`) or in *backward* (`<<`) direction:

[17]	*step*	::=	(**>>** | **<<**) *axis* [*anchor*]
[18]	*axis*	::=	**types** | **supertypes** | **players** | **roles** | **characteristics** | **scope** | **locators** | **indicators** | **reifier** | **atomify**

The optional anchor adds control information which is useful with some axes, but no others. If it is missing, `tm:subject` will be assumed.

When the anchor is evaluated, it must evaluate to a topic item and is interpreted as type. Then in all navigation steps, type

transitivity is honored, i.e. not only the identified item is considered but also all of its subtypes.

Given a map and a single value (be it an atom or an item in the map), the following axes are defined:

types

In forward direction this step computes all *types* of the value according to 3.3. In backward direction this step produces all *instances* of the value. The optional item has no relevance.

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

Additional Notation: Asking for all instances of an item is the inverse of asking for types.

[B]	*step*	::=	**>>**	**instances**
		==>	**<<**	**types**

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

supertypes

In forward direction this step computes all *supertypes* of the value according to 3.3. In backward direction this step produces all *subtypes* of the value. The optional item has no relevance.

EXAMPLE:
`person >> supertypes` produces all supertypes of the concept `person`, say, `human`, `mammal`, etc. depending on the used 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 specifies the type of the roles to be considered whereby class transitivity is respected. If a playing topic plays several roles in such an association item, then it appears as many times in the result (multiset interpretation).

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 plays several roles in one and the same association, this association will appear as many times. The optional item specifies the type of the roles to be considered whereby type transitivity is respected.

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

[C]	*step*	::=	**->**	*anchor*	
		==>	**>>**	**players**	*anchor*
[D]	*step*	::=	**<-**	*anchor*	
		==>	**<<**	**players**	*anchor*

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 uses of the same role type in one association causes multiple results. The optional item has no relevance.

If the value is a topic item, in backward direction this step computes all association items where that topic is the role type. Multiple uses of one topic as role in one association causes 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 names and occurrences of that topic which are (direct or indirect) subtypes of the item specified. The result is a sequence of name and occurrence items, not the atomic values in these.

If the value is a name or an occurrence item, in backward direction this step computes the topic to which the name or the occurrence is attached. The optional item can be used to control the type of the items one is interested in. Also here type transitivity is honored.

EXAMPLE:
The following navigation finds all occurrences of `jack`:

```
jack >> characteristics tm:occurrence
```

EXAMPLE:
The following navigation finds all nicknames of `jack`:

```
jack >> characteristics nickname
```

EXAMPLE:
The following navigation finds all names and occurrences of `jack`:

```
jack >> characteristics *
```

scope

In forward direction, this navigation leads from characteristics (names and occurrences) and association items to

their scope.

In backward direction, this navigation leads from a topic to all associations and characteristic items in that scope. The optional item has no relevance.

Additional Notation: To extract scoping information the following shorthand can be used:

```
[E]  step                  ::=  @
                           ==>  >>  scope
```

locators

If the value is a topic item, in forward direction this step retrieves all subject locators (subject addresses) of this item. If the value is a IRI, in backward direction this step retrieves all topic items which have this IRI as subject locator. The optional item has no relevance.

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

```
[F]  step                  ::=  =
                           ==>  <<  locators
```

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, but only if a topic with that subject locator exists in the map. If not, the result will be empty.

indicators

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

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

```
[G]  step                  ::=  ~
                           ==>  <<  indicators
```

EXAMPLE:
In a map about dictators the expression `http://en.wikipedia.org/wiki/Stalin ~` would indicate the subject *Stalin* and would return a topic item if such a topic existed in the map with the IRI as subject indicator.

EXAMPLE:
The expression `file:/maps/dictators.xtm ~` indicates the subject which is the map. If a topic with such an subject identifier exists in the map, that topic item is the result. Otherwise, the result is empty.

This is in contrast to using directly a item reference (4.3), such as `file:/maps/dictators.xtm` (without the quotes and without the navigation). In that case it is an error if no such topic exists with that subject identifier.

reifier

If the value is a topic item, then in forward direction this steps finds the association, name or occurrence item which is reified by this topic. If the topic reifies a map, then all items in that map will be returned and the context map `%_` will be set to this map for the remainder of the directly enclosing TMQL expression. The optional item has no relevance.

If the value is an association, a name or an ocurrence item, then in backward direction this step finds any reifying topic.

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

```
[H]  step                  ::=  ~~>
                           ==>  >>  reifier
```

EXAMPLE:
If the topic `stalin-dictatorship` would reify an association where `stalin` plays the role `dictator`, then the expression `stalin-dictatorship ~~> >> players dictator` would render the topic item `stalin`.

EXAMPLE:
To find all items in the map stored in the file `/maps/dictators.xtm` the expression `file:/maps/dictators.xtm ~~~>` can be used.

atomify

If the value is a name or occurrence 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 it. The item is effectively converted to an atom according to the atomification rules (4.5). The optional item has no relevance.

If the value is an atom, in backward direction this step *de-atomifies* immediately the atom and returns all names and occurrences where this atom is used as data value. Also here the optional item has no relevance.

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` occurrences with a certain URL (they cannot be names as these cannot contain URLs):

```
"http://myhomepages/jack" << atomify
                          << characteristics homepage
```

Additional Notation: If topic characteristics should be automatically atomified, the following shorthand can be used:

```
[I]  navigation            ::=  /  anchor  [ navigation ]
                           ==>  >>  characteristics  anchor  >>  atomify  [ navigation
                                ]
```

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

```
$p / homepage
```

Additional Notation: If atomic values should be automatically looked up in characteristic items of a certain type, the following shorthand can be used:

```
[J]   navigation          ::=   \ anchor [ navigation ]
                          ==> <<  atomify  <<  characteristics  anchor [ navigation
                              ┘
```

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

```
23 \ age
```

EXAMPLE:
The expression "Stalin" \ name computes all topic items which have the name "Stalin".

In all combinations not listed above the atomification is the identity function.

NOTE:
Implementations can redefine the semantics of the atomification/deatomification function. This can be used, for instance, to convert between topic items and application-specific objects to be handled by the calling application.

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

4.5 Auto-Atomification

Topic names and occurrences are experienced in an ambivalent way: either as items, including not only the data value, but also the scope and the type of the item; or the data value only.

When such item is subjected to an *atomify* navigation step, the atomic value is not immediately extracted, but the process has to be postponed. If the atomification would be done immediately, the information for scope (and type) would be lost for subsequent processing steps.

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

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

Atomification is postponed until one of the following situations occur:

1. The name or occurrence is about to be passed to the environment as part of the result.
   EXAMPLE:
   The following query expression will return the homepage URLs as IRIs (and not occurrence items):
   ```
   select $p / homepage
   where
      $p isa person
   ```
2. The name or occurrence is about to be compared in the process of ordering (asc or desc, 4.8.2).
   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
   ```
3. The name or occurrence is about to be passed into a function in the process of a function invocation (4.12).
   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
   ```
   More specifically, also when the name or occurrence is about to be compared to an atomic value.
   EXAMPLE:
   The following query expression will return all persons younger than 42:
   ```
   select $p
   where
      $p / age < 42
   ```
4. The name or occurrence is about to be passed into a predicate in the process of predicate invocation (6.6.4).
   EXAMPLE:
   The following query expression will return all persons younger than methusalem:
   ```
   select $p
   where
      $p / age < methusalem / age &
      $p isa person
   ```
5. The name or occurrence is used inside an XML fragment.
   EXAMPLE:
   In the following, the names of each book will be atomified so that they can be inserted as text into the XML stream:

```
return
   <books>{
    for $b in // book
    return
       <title>{ $b / name }</title>
  }</books>
```

6. The name or occurrence is used inside a TM fragment, except when used on positions where a name or occurrence declaration is expected ([CTM]).

   EXAMPLE:
   The following query expression will return a TM fragment with persons from the context map who are younger than `methusalem`. All of these persons get the `homepage` characteristics copied verbatim, their name(s) is (are) embedded into the newly generated `comment` characteristics:

```
for $p in // person
where
   $p / age < methusalem / age
return """

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

"""
```

7. The name or occurrence item is subjected to a *de-atomification* step.

   EXAMPLE:
   In the expression `$book / name [ @ english ] \ title` a book item is first used to extract the names. These are then filtered for english versions only. To de-atomify the value, it is first extracted from the remaining names. Only then all names and occurrences of type `title` are generated.

If a name or occurrence item is *not* scheduled for atomification, it will remain an item.

EXAMPLE:
The following query expression will return the `homepage` occurrences themselves, not the values within them.

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

4.6 Simple Content

A simple value (*anchor*) is either provided as constant or as value of a variable:

[19]	*anchor*	::=	*constant* | *variable*

Simple content is formed by an anchor followed by any number of navigation steps:

[20]	*simple-content*	::=	*anchor* [*navigation*]
[21]	*navigation*	::=	*step* [*navigation*]

First the anchor is evaluated in the current context. The resulting tuple sequence will then be subjected to any navigation step in lexical order. The result of the last step is the overall result of the simple content.

EXAMPLE:
The simple content `jack >> occurrence >> types` computes all types of all occurrence a topic with item identifier `jack`.

EXAMPLE:
The simple content `$t / nickname` computes all nicknames of a topic stored in the variable `$t`.

4.7 Composite Content

When a query expression is evaluated, it will (on successful termination) generate *content*. That always has the structure of a *tuple sequence*, i.e. a sequence of tuples consisting of atomic values, be they literal values such as integers or strings, or be they Topic Map items or XML fragments.

Content can be constructed in different ways:

[22]	*content*	::=	*content* (**++** | **--** | **==**) *content* | **{** *query-expression* **}** | **if** *path-expression* **then** *content* [**else** *content*] | *tm-content* | *xml-content*

1. When the binary infix operators `++`, `--` or `==` are used, `--` and `++` are interpreted from left-to-right (left-associative). The operator `==` has the highest precedence.
2. Content can further be generated unconditionally with a nested query expression or conditionally with an if-then-else construct. If the ELSE branch is missing, then `else ()` will be assumed.
3. Content can also be constructed as Topic Maps fragment (4.10) or as XML fragment (4.9).

1. If content is combined with one of the binary operators, first both content 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 (*AND semantics*). Tuples are compared according to 4.8.1.

      EXAMPLE:
      In the expression `%map [ . >> types == person ]` inside the filter each item from the `%map` is tested whether one of its types is `person`.

      In order to achieve this, the map, a tuple sequence itself, will be iterated through. Inside the filter the first component of every individual tuple is extracted. For this the list of types is computed. That resulting tuple sequence is then compared with one containing only one singleton tuple with the item for `person`.

      EXAMPLE:

The following query expression lists all the person's 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 (*OR semantics*).

   NOTE:
   The overall result sequence may contain duplicates.

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 (*except semantics*). Tuples are compared according to 4.8.1.

   EXAMPLE:
   To following tuple expression can be used to find all evil people in the universe:

```
// person -- // evil-evildoer
```

2. Content can also be unconditionally generated with a query expression (Clause 6) when that is wrapped inside `{}` brackets.
3. When a content is conditional (if-then-else), the result depends on the condition path expression. If that is evaluated (6.6) in the current context and if that results in at least one tuple, then the overall result of the conditional is that of the content in the `then` branch, otherwise that of the `else` branch.

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

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

Additional Notation: When the content can be produced by a simple path expression, then the curly brackets can be dropped:

[K]	*content*	::=	*path expression*
		==>	**{** *path expression* **}**

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

[L]	*content*	::=	*path-expression-1* **||** *path-expression-2*
		==>	**if** *path-expression-1* **then** **{** *path-expression-1* **}** **else** **{** *path-expression-2* **}**

EXAMPLE:
The following expression selects all `person` names. If a person does not have a name, then the special value `undef` will be used:

```
select $p / name || undef
where
   $p isa person
```

4.8 Tuples and Tuple Sequences

4.8.1 Tuples

Tuples are ordered collections of simple values (atoms and items). The length of a tuple is the arity of the collection. The components of a tuple can be of different types.

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

As individual values of a tuple are ordered, the first value is assigned the index `0`, the next `1`, etc. *Projection* (6.6.3) can be used to extract one (or more) values from a given tuple. If a projection refers to an index larger or equal the length of a tuple, the extraction will result in an error.

NOTE:
Tuples cannot be directly denoted, only via tuple expressions. These produce zero, one or more tuples.

EXAMPLE:
The tuple expression `(42, "DONT PANIC")` will always produce a single tuple with the integer value `42` at index `0` and the string `DONT PANIC` at index `1`.

4.8.2 Comparing Tuples

Tuples are only then equivalent, if they have the same length and all the values with same index are equivalent according to the equality rules of their type.

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` (for *ascending*) or `desc` (for *descending*). 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 starting with index `0`.

   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 length is also

the smaller tuple.

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. If the values cannot be compared, then the ordering is undefined.

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)`.

4.8.3 Tuple Expressions

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

[23]	*tuple-expression*	::=	**(** < *value-expression* [**asc** | **desc**] > **)**

Each column contains a value expression, optionally followed by an ordering direction.

When a tuple expression is evaluated, 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.

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.

A *non-empty tuple sequence* is one which contains at least one tuple. The *empty tuple sequence* does not contain a single tuple.

Additional Notation: The constant `null` represents the empty tuple sequence. It is typeless per se and is used for situations when no particular value (also not `undef`) should or can be used.

[M]	*tuple-expression*	::=	**null**
		==>	**(** **)**

4.8.4 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 these 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 as defined in 4.8.2 is used.

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

   EXAMPLE:
   The following expression finds first all instances of `person`; then for each of these, the combinations of their `name`s and `age`s are generated:

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

   The second component of the projection carries an order direction, so that of the first component will default to `asc`. The resulting tuple sequence will then be ordered, first according to the name value; when there is a draw, then according to the age information, but that in descendant order. Note, that this is done for each person independently.

2. If the tuple sequence is generated from two tuple expressions, *TS1* and *TS2*, via the binary operator `++` 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 expression will return a list of `person` names:

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

   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 expression also selects a list of names, like the query above. This time, however, the *partial* lists of names for a single person is sorted by the name. Still the individual blocks of names are sorted by the `person`'s ages.

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

4.8.5 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, separated by commata.

When a tuple sequence is *stringified* then the string representations of the individual tuples will be concatenated, separated by a single carriage-return character (`#x0D`). If the tuple sequence is ordered, this order is also carried over.

4.9 XML Content

XML content follows a subset of the syntactic rules given in the XML specification [XML 1.0] with one notable extension: XML content can contain query expressions (Clause 6) to generate flexible output. These expressions must be properly nested using a balanced pair of curly brackets (`{ }`).

[24]	*xml-content*	::=	{ *xml-element* }
[25]	*xml-element*	::=	**<** *xml-tag* { *xml-attribute* } *xml-rest*
[26]	*xml-tag*	::=	[*prefix*] *xml-fragments*
[27]	*xml-attribute*	::=	[*prefix*] *xml-fragments* **=** **"** *xml-fragments* **"**
[28]	*xml-rest*	::=	**>** { *xml-element* | *xml-fragments* } **</** *xml-tag* **>** | **/>**
[29]	*xml-fragments*	::=	{ *xml-text* | **{** *query-expression* **}** }
[30]	*xml-text*	::=	...see text...

Between the prefix and the following tag name or attribute name there must not be any whitespace.Text within attribute values may not include the string terminator `"`. XML text within an element may not include terminators `<`, `>`, `{` or `}`. If the characters `{` and `}` are used as-is, they have to be encoded as `&#007B;` and `&#007D;`, respectively.

EXAMPLE:
The following XML content samples are valid:

- `<copyright>Copyright Holder</copyright>`
- `<message>Celine Dion has quite a different sound than {$bn}.</message>`
- `<message{$kind} title="{$x}">This may work.</message{$kind}>`
- `<pro:code xmlns="{$uri}" lang="pascal">procedure TEST () &#007B; writeln; &#007D;.</code>`

The following XML content samples are invalid:

- `<copyright>Copyright Holder` (no end tag)
- `<code lang="pascal">procedure TEST () { writeln; &#007D;.</code>` (opening `{` indicates subexpression)

The following XML content may produce run-time errors:

- `<mess{$x}>This is broken.</{$y}age>` (XML wellformedness depends on variable binding)

NOTE:
There is no support for XML processing instructions or CDATA segments, DOCTYPE constructs and XML comments.

Within an XML text stream whitespaces are significant, also those which precede the opening tag and those which follow the closing tag.

EXAMPLE:
In the following FLWR expression the nested `<person>` element is preceded by a line-break and 3 blanks and is followed by a line-break. For each iteration over `person` instances these whitespace characters must be added to the result XML fragment:

```
return
  <persons>{
    for $p in // person
    return
       <person>{$p / name}</person>
  }</persons>
```

There is also a line break before the `<persons>` opening tag which will be part of the overall result fragment.

When XML content is evaluated, first all nested query expressions are evaluated (in no defined order) in the current context. The content generated by these expressions is then embedded into the XML content, replacing the text of the query expressions (including the `{}` bracket pair) according to the following embedding rules:

- Atomic content is converted into its string representation.
- String content is used as-is except that special characters `&`, `"`, `<`, `>`, `'` are automatically encoded to the predefined entities `&`, `&quot;`, `<`, `>`, `&apos;`, respectively. String content is not delimited by quotes.
- XML content is used as-is.
- Tuple sequences are converted into their string representation (4.8.5).
- TM content is serialized using XTM ([XTM 2.0]).

The resulting text is interpreted as XML content which matches the *element* in [XML 1.0]. It is an error if XML parsing fails.

EXAMPLE:
Given the following code,

```
for $s in "Portishead",
    $x in <some>XML code</some>
return
   <message>{$s} has no idea about {$x}.</message>
```

will return the XML fragment `<message>Portishead has no idea about <some>XML code</some>.</message>.`

4.10 Topic Map Content

TM content is constructed using CTM [CTM]:

[31]	*tm-content*	::=	**"""** *ctm-instance* **"""**

Additional 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 and every association item will be injected into the CTM instance. Every string with identifier syntax will be interpreted as item identifier and a topic with such item

identifier will be injected into the CTM instance. Every string which follows the IRI syntax will be interpreted as subject identifier; also here a topic will be injected into the CTM instance, using this very subject identifier.

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 embedded 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 its item identifier isused.

EXAMPLE:
In the following query expression for each item of class `person` a topic will be declared and an association is added which records the fact that that person is employed:

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

   {$p ~ }                    # find subject identifier
   {$p / name}                # add all names
   homepage: http://company.org/{$p}.html

   # add association
   is-employed-at ( employer: bigcorp, employee: {$p} )

"""
```

The item identifier of a person item in the queried map will also become the item identifier in the generated map. The name characteristics are copied into the generated map. A new occurrence of type `homepage` is then added with the generic URL parameterized by the item identifier of the topic.

4.11 Value Expressions

Value expressions are expressions which produce a single value, or a sequence of values:

[32]	*value-expression*	::=	*value-expression* *infix-operator* *value-expression* | *prefix-operator* *value-expression* | *function-invocation* | *content*
[33]	*infix-operator*	::=	...any in the predefined environment...
[34]	*prefix-operator*	::=	...any in the predefined environment...

Value expressions are either directly producing content, using the invocation of a function, 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 Annex A. Every function there marked as *infix* can be used as infix operator, every function marked there as *prefix* can be used as prefix operator. All operators are eventually mapped into their function equivalent, so that a value expression either generates content directly or computes it via a function application.

EXAMPLE:
The following are valid value expressions:

```
1 + 2
$person / age
$person / age >= 18
```

4.12 Function Invocation

[35]	*function-invocation*	::=	*item-reference* *parameters*
[36]	*parameters*	::=	*tuple-expression* | **(** < *identifier* **:** *value-expression* > **)**

A function is addressed via an item reference which will be resolved in the effective map (6.2). It is an error if no such function exists there.

Every function must have formal parameters to which the actual parameters will be assigned. How the formal parameters are made known to the TMQL processor is not specified by this International Standard. The formal parameters can be *positional*; in this case their name is `$0`, `$1`, etc. Or they can be explicitly *named*.

1. For a *positional parameter association*, first the tuple expression is evaluated in the current context rendering a tuple sequence. One by one, a tuple from this sequence is taken. The individual values of that tuple are then assigned to the formal parameters `$0`, `$1`, etc. It is an error if there are less formal parameters than values in the tuple. Then the function is invoked.

   The individual function results for every tuple form the result tuple sequence. Any ordering of the parameter tuple sequence will be maintained in the result.

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

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

   Should a person have several `age` occurrences, then the result list will contain the square root for each of these.
2. For a *named parameter association*, first all value expressions are evaluated in the current context. Also here the result is effectively a tuple sequence except that the components of individual tuples are not addressable via an index, but a name.

   For each such (named) tuple the function is invoked, whereby actual and formal parameters are associated via their name. While it is allowed 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.

   EXAMPLE:
   A function `nr-accounts` would be called as:

   ```
   nr-accounts (owner: "James Bond")
   ```
3. In addition to the formal parameters, the effective map is passed as a special parameter to the function. How this is accomplished is not specified by this International Standard.

4.13 Boolean Expressions

4.13.1 Structure

WHERE clauses (6.4, 6.5) and filters (6.6.2) make use of boolean expressions:

```
[37]  boolean-expression   ::=  boolean-expression | boolean-expression   |
                                 boolean-expression & boolean-expression   |
                                 boolean-primitive
[38]  boolean-primitive    ::=  ( boolean-expression )  |
                                 not boolean-primitive  |
                                 forall-clause          |
                                 exists-clause
```

Boolean expressions can be combined with the binary boolean operators `&` (AND) and `|` (OR). Boolean primitives can be negated. The brackets `()` can be used to override the usual precedence, namely that `&` binds stronger than `|` and that `not` binds stronger than `&`. 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.

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 subsequent `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 ...
```

NOTE:
As the language is ontological neutral, the constants `true` and `false` have no semantic meaning. They are just values, like all integers or all strings. Any (simple or complex) value can play the role of the traditional *true*, and the empty tuple sequence plays the role of *false*.

Before a boolean expression is evaluated all free occurrences of the variable `$_` are *existentially quantified* according to 5.3. Then the boolean expression is evaluated in the current context.

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.

If the evaluation of a boolean primitive renders a non-empty tuple sequence, then negating it with `not` will return the empty tuple sequence. Otherwise, an arbitrary non-empty tuple sequence will be generated.

The operator `&` represents the logical AND; and `|` the logical OR.

Additional Notation: To test whether the value of `$0` (the first component of the current tuple) is of a particular type or in a particular scope, the following shorthands are provided:

```
[N]  boolean-primitive   ::=  ^ anchor
                         ==>  . >> types == anchor
[O]  boolean-primitive   ::=  @ anchor
                         ==>  . @ == anchor
```

EXAMPLE:
To select only the english names from `person`s, one can use:

```
// person / name [ @ english ]
```

4.13.2 EXISTS Clauses

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

```
[39]  exists-clause      ::=  exists-quantifier binding-set satisfies boolean-expression
[40]  exists-quantifier  ::=  some | at least integer | at most integer
```

If the EXISTS clause is introduced with the token `some`, then it is an *numerically unrestricted EXISTS clause*. Otherwise it is called *numerically restricted*. For these, the integer must be positive. In the case of `at least` that integer is a lower bound, for `at most` it is an upper bound.

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

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

First, a sequence of binding sets is generated. Then each of these sets will be pushed onto the current context for the evaluation of the boolean expression.

A numerically unrestricted EXISTS clause evaluates exactly then to a non-empty tuple sequence if there exists at least one such binding set for which the evaluation of the boolean expression returns a non-empty sequence. If there is no such binding (or no binding at all), the overall result is the empty sequence.

EXAMPLE:
The following expressions all return the empty sequence:

```
some $a in null satisfies exists 1     # there is no value in null

some            satisfies exists 1     # not a single binding

some $a in %_   satisfies null         # many values, none satisfies
```

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

[P]	*exists-clause*	::=	**exists** *content*
		==>	**some** *$_* **in** *content* **satisfies** **not** **null**

EXAMPLE:
The following boolean expression tests whether a given `person` item (bound to `$person`) is an author, i.e. is involved in an association via the role `author`:

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

Additional Notation: The syntax allows the `exists` token to be omitted:

[Q]	*exists-clause*	::=	*content*
		==>	**exists** *content*

EXAMPLE:
That the `exists` token can be omitted, allows to write more intuitively

```
select $p / name
where
  $p isa person
```

instead of the more canonical

```
select $p / name
where
  exists $p isa person
```

A numerically restricted EXISTS clause with a lower bound *N* evaluates exactly then to a non-empty tuple sequence if there exists at least *N* (including *N*, or more) such binding sets for which the evaluation of the boolean expression returns a non-empty sequence. If there is no such binding set (or no binding set at all), the overall result is the empty sequence.

EXAMPLE:
The following tests whether there are at least 3 persons over nineteen

```
at least 3 $p in // person
satisfies
    $p / age >= 19
```

A numerically restricted EXISTS clause with an upper bound *N* evaluates exactly then to a non-empty tuple sequence if there exists at most *N* (including *N*, or less) such binding sets for which the evaluation of the boolean expression returns a non-empty sequence. If there is no such binding set (or no binding set at all), the overall result is a non-empty sequence.

4.13.3 FORALL Clauses

A FORALL clause can be used to test whether all tuples from a given tuple sequence satisfy a certain condition:

[41]	*forall-clause*	::=	**every** *binding-set* **satisfies** *boolean-expression*

Like for EXISTS clauses, a sequence of bindings is generated first. Each of these bindings is pushed onto the context for the evaluation of the boolean expression.

A FORALL clause is evaluated in the current context. It evaluates to a non-empty tuple sequence if for all generated binding sets the boolean expression evaluates to a non-empty tuple sequence. Otherwise it will evaluate to the empty sequence. It also evaluates to a non-empty sequence if not a single binding set could be generated in the first place.

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

```
every $p in // politician satisfies
      $p <- person -> trait == honorable
```

If our universe would not contain a single politician, then this boolean expression would evaluate to a non-empty sequence. Otherwise, only when each of them has (at least one) trait with value `honorable`, only then the expression evaluates to a non-empty sequence.

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

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

NOTE:
TMQL is assuming a *closed world*, so there is a semantic relationship between EXISTS and FORALL clauses:

[R]	*boolean-expression*	::=	**every** *binding-set* **satisfies** *boolean-expression*
		==>	**not** **some** *binding-set* **satisfies** **not** **(** *boolean-expression* **)**

5 Query Contexts

5.1 Variables

During evaluation variables are used to bind values. To identify a particular variable, a variable identifier is used. It must be prefixed by a *sigil* and can be postfixed by any number of primes (`'`):

[42]	*variable*	::=	**/[$@%][\w#]+'*/**

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

EXAMPLE:
Valid variables are `$a`, `$a'`, `$_`, `@a_long_list_name` or `$23`. Examples for invalid variables are `x` (sigil missing), `$a-string-world` (dashes not allowed in names) or `@list '` (no blanks before the prime are allowed).

Following special variables are assigned automatically during an evaluation. They cannot be redefined in variable assignments (5.4).

`%` (current environment map)
The environment map (see 6.3) contains all necessary background knowledge for a TMQL processor. This includes all data types and their related functions from the *predefined environment* (Annex A).

It is initially adopted from the querying application (explicitly or implicitly). Query expressions can locally enrich this environment by adding functions, predicates or ontologies.

`%_` (current context map)
Whenever a map has been referred to (as in a FROM clause within a SELECT query expression, via a path expression, or via a reification navigation step), it becomes the current context map. All item references 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*.

`$_` (anonymous variable)
This write-only variable can be used as placeholder inside a boolean expression if the value it binds to is not of interest to the result.

`$#` (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()` (Annex A).

`$0`, `$1`, `$2`, ... (positional variables)
Whenever one particular tuple (sequence) is considered, `$0` projects the first column from it. `$1` projects the second, `$2` the third and so forth.

Additional Notation: As it appears quite frequently that the first (and often only) component of a tuple is to be addressed, we introduce a shorthand:

[S]	*variable*	::=	**.**
		==>	**$0**

EXAMPLE:
To select only persons who are older than `18`, one can use `// person [. / age > 18 ]`.

Variables always exist in a *lexical scope*, i.e. a lexical part of a query expression. For variables which are assigned within a variable assignment (5.4) this scope starts directly after the assignment and reaches until the syntactic end of the directly enclosing query expression. For all other variables the scope is the whole query expression. These *global variables* cannot be rebound to other values within the query expression; they are effectively treated as constants.

5.2 Variable Bindings

A *variable binding* connects one particular variable with a value. A *binding set* is a set of such bindings, with the constraint that one particular variable may only appear once.

EXAMPLE:
The set `{ $a => 23, $b => "text" }` is a binding set.

Once a variable is bound to a particular value, this binding cannot be changed. The same variable can get a different value in another binding, though, hiding the former binding (immutability of variables).

During the course of the (nested) evaluation of a query expression, a processor will maintain stack of binding sets, the *variable context* (short: context).

The *value of a particular variable* in the context is determined by a binding for that variable in that binding set which has been added last to the context.

A processor will always maintain the following constraints on contexts:

1. If the variable names differ only in the number of primes, then their values MUST differ.
2. Any two different variables may be bound to different or the same values. They are regarded to be independent.

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

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

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

```
...
where
    is-neighbor-of (* : $a, * : $b)
  & is-neighbor-of (* : $a, * : $c)
  & is-neighbor-of (* : $b, * : $c)
```

5.3 Implicit Existential Quantification

Like any other variable, the anonymous variable `$_` can be used as placeholder to bind to any value. It is, however, *write-only* in that its value can never be retrieved; any occurrence of `$_` (regardless whether in the same scope or not) is effectively a different variable. This is useful when one is not interested in the actual value(s) a `$_` occurrence binds to, but only the fact that it actually does bind.

EXAMPLE:
The following query expression finds all person's name(s) who live in a city; which city it is, is not relevant:

```
select $p / name
where
   $p isa person
 & lives-in-city (being : $p , city : $_)
```

Whenever a free anonymous variable is used, a processor will implicitly let it range over all map items, such that the boolean expression can be satisfied. As such, every occurrence of `$_` is *implicitly existentially quantified* using a new, unique (internal) variable name.

EXAMPLE:
The WHERE clause above equivalently can be written as:

```
where
   $p isa person
 & some $_273 in // * satisfies
        lives-in-city (being : $p , city : $_273)
```

NOTE:
Anonymous variables cannot bind to atomic values.

EXAMPLE:
The following query expression is valid, but will never return a reasonable result:

```
select $p / name
where
   $p / age > $_
```

5.4 Variable Assignments

New variable bindings can be created during the course of a query evaluation. In general, with variable assignments a *sequence of bindings* is generated:

[43] *variable-assignment* ::= *variable* **in** *content*

The evaluation of the content will result in a tuple sequence. How many different variable bindings are produced by such an assignment then depends on the variable sigil:

1. If the variable sigil is `$`, so that the variable can only be bound to 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 accordingly; also every tuple will be iterated through following increasing indices.
2. If the variable sigil is `@`, so that the variable can only hold tuples, then — one by one — the tuples within the tuple sequence are bound to the variable for individual bindings. If the tuple sequence is ordered, the bindings will follow this order.
3. If the variable sigil is `%` so that the variable can hold a complete tuple sequence, then the complete tuple sequence will be bound; only one binding will be generated.

EXAMPLE:
The following will create as many bindings as there are `person` topics in the context map. Every binding contains `$p` bound to one `person` item.

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

EXAMPLE:
The following will create a single tuple of all `person` items. The function `fn:concat` takes a tuple sequence and produces one which consists only of the concatenation of all the original tuples.

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

As a generalization, several variables can be assigned simultaneously to create a sequence of binding sets:

[44] *binding-set* ::= < *variable-assignment* >

If the list itself is empty, or no bindings can be generated for some variable, then a binding set may be empty.

EXAMPLE:
The following creates a sequence of binding set where each of these has a binding for `$p` and `$c`. Their respective values are topic items; `$p` iterates over all `person` items and `$c` iterates over all cities, so that every combination is generated.

```
$p in // person , $c in // city
```

5.5 Binding Set Ordering

Sequences of binding sets can be ordered according to an *ordering tuple expression*.

If the ordering tuple expression is empty, then no ordering will occur.

For this purpose, the ordering tuple expression will be evaluated separately for every binding set in the current context. If the resulting tuple expression does not contain a single tuple, the value `null` is used. If the result contains more than one tuple, any of these can be used.

A binding set *B1* is said to *occur before* another, *B2*, if *B1*'s (single) tuple is *smaller* than that of *B2* according to the tuple comparison rules (4.8.2).

EXAMPLE:
The following FLWR expression creates a sequence of binding sets where in each of them `$p` will bind a particular item of type `person`. This is sequence is ordered according to the values of `$p / name`. In this order the binding set is used to evaluate the rest of the expression.

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

6 Query Expressions

6.1 Processing Model

Every query expression is evaluated in a context (5.4). When an evaluation of a query expression is initiated, the application (direct or indirectly) will pass in an *initial binding set*. How this is achieved is not constrained by this International Standard. This binding set forms the initial context (5.4). Apart from this no other information is imported.

The only variable which MUST be defined in the initial binding set is `%%` (environment map, 5.1, 6.3). If a particular topic map to be queried should be passed in, it can be bound to the context map `%_`, so that is used by default.

During processing, further binding sets are created and are pushed onto the context for the duration of the evaluation of any subexpressions. After that, the last binding set added is removed (popped) from the context.

During an expression evaluation erroneous situations may arise, at the detection of which the processor must terminate processing. It remains unspecified how processors signal this to the processing environment.

Query evaluations return results into the environment. 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).

6.2 Structure

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

[45]	*query-expression*	::=	[*environment-clause*] (*select-expression* | *flwr-expression* | *path-expression*)

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, but only FLWR style queries can generate XML and TM content. Query expressions can be nested in several ways and it is also possible to mix the different styles within one larger expression.

The *environment clause* allows to declare and import additional (ontological) knowledge such as predicates and functions into the querying process (6.3), further to that of the predefined environment (Annex A).

First, the environmental clause will be evaluated in the current context. The resulting map will be merged with the current environment map and will be bound to a new instance of the variable `%%`. This binding will be added locally into the current context with which the rest of the query is evaluated.

The *effective map* to be queried is the merge of the following:

- the context map (the current value of `%_`)
- the environment map (the current value of `%%`) of the directly enclosing query expression, or that which the caller provided

6.3 Environment Clause

The environment map contains background knowledge a processor will use in addition to the context map. In many cases 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; and
- functional dependencies (*functions*); and
- general ontological constraints and rules (*predicates*); and
- external ontologies (*vocabularies*, *taxonomies*, *PSI sets*).

How these objects are defined within an ENVIRONMENT clause is itself not syntactically constrained by this International Standard, except that triple-quotes have to be used as enclosure:

[46]	*environment-clause*	::=	""" ... """

Irrespective of the notation, a processor has to register every such object in the *environment map*:

1. For every function declared in the clause a topic of type `tmql:function` has to be registered. A function can then be addressed via an *item-reference* (4.3) for that topic the environment map. Information about the parameter passing and the returning of values is not constrained by this International Standard.
   NOTE:
   Implementation may add any additional information for this function, such as description, names or a parameter profile. Implementation may also add the source code of the function as occurrence values, using different scopes for different implementation languages.
2. For any ontological rule a processor has to register a topic of type `tmql:predicate`. A predicate is addressed via an *item-reference* (4.3) for that topic in the environment map. The information about the parameter association is not constrained by this International Standard.

3. Any external vocabulary, additional ontologies, or prefixes for namespaces have to be registered in the environment map as topics of type `tmql:ontology`. The item identifier of such topics will act as *prefix* which can be used in QNames (4.2).

NOTE:
The lexical scope of functions, predicates and ontology prefixes derives from that of the environment map. Consequently, any concept declared in an environment map can be used inside the directly enclosing query expression and all its subexpressions.

NOTE:
All of the above objects are represented by topics and the environment map can be also queried separately. This mechanism enables reflection.

EXAMPLE:
To find all functions (and their description) one can use:

```
select $f / name, $f / description
from %%
where
    $f isa tmql:function
```

6.4 SELECT Expressions

Query expressions can take the form of SELECT expressions:

[47] *select-expression* ::= **select** < *value-expression* >
[**from** *value-expression*]
[**where** *boolean-expression*]
[**order** **by** < *value-expression* >]
[**unique**]
[**offset** *value-expression*]
[**limit** *value-expression*]

Only the SELECT clause is mandatory. 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 not null` 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 none of the values bound to it will 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. If there was no LIMIT clause, `$_limit` remains unbound.

Then all free unbound variables in the WHERE clause (not those in the SELECT clause, and not any occurrences of `$_`) are determined. Each of these variables will be *existentially 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.

NOTE:
According to this, query expressions which name variables in the SELECT clause which are not yet bound by the environment and do not constrain them inside the WHERE clause are erroneous. This is meant as a safeguard to avoid that queries accidentially traverse — potentially huge — topic maps. This is also the motivation to rule out the use of anonymous variables there.

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 tm:subject
```

One by one, every such binding set will be added to the current context whereby the variable semantics (5.2) is honored. If no ORDER clause exists, then this sequence of contexts will remain unordered. Otherwise it will be sorted according to 5.5 using the expression(s) in the ORDER clause.

EXAMPLE:
In the following query expression the result will be all names of all instances 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 (4.13). If the evaluation result is not the empty sequence, then that particular context will be used to evaluate the tuple expression within the SELECT clause. Otherwise, this particular context will be ignored.

All the evaluation results of the SELECT clause are concatenated into one tuple sequence. If the sequence of contexts was unordered, so will be that tuple sequence, otherwise this concatenation will be ordered accordingly, whereby any existing ordering inside the partial subsequences will remain.

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

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

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 expressions allow not only to generate tuple sequences to be returned, but also the construction of content in XML and TM form:

[48]	*flwr-expression*	::=	[**for** *binding-set*] [**where** *boolean-expression*] [**order** **by** < *value-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 not null` is assumed. If the FOR clause is missing `for $_ in 1` is assumed.

A given FLWR expression can contain any number of *FOR clauses*. Several variable associations can be listed in one particular *FOR clause*. This is equivalent to having a dedicated FOR clause for every individual variable. Any such introduced variable is visible until the end of this FLWR expression.

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

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

EXAMPLE:
The following FLWR expression returns all pairs of (different) `person` topic items where the `person`s are members within the same `group`. The `group` itself is ignored:

```
for $p in // person
  for $p' in // person   # Note: $p is never the same as $p'
  where                 #       so there is no tautology
     $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 will be subjected to a test provided by the boolean expression in the WHERE clause. If the evaluation renders a non-empty sequence, then that context will be kept; otherwise it will be discarded.

For each kept context the RETURN clause will be evaluated. Each of these evaluations will result in a sequence of tuples. If — due to enclosing FOR clauses — several results have been computed for every iteration, then these partial results are combined into a using the operator `++`. This forms the overall result for the FLWR query expression.

6.6 Path Expressions

6.6.1 Structure

Path expressions follow a *navigate and filter* approach. Starting from given values (atoms or items in a map), navigation steps along defined axes within the context map compute new values. These values then can be filtered according to boolean conditions or these values can be used as new starting points.

As starting point either simple content or a whole tuple sequence (4.8.3) can be used. This is then followed by any number of postfixes.

[49]	*path-expression*	::=	*postfixed-expression* | *predicate-invocation*
[50]	*postfixed-expression*	::=	(*tuple-expression* | *simple-content*) { *postfix* }
[51]	*postfix*	::=	*filter-postfix* | *projection-postfix*

The evaluation of the tuple expression (4.8.3) or simple content (4.6) always results in a tuple sequence. The navigational aspect of path expressions is provided by the simple content. *Filter postfixes* simply remove all tuples from the sequence which do not satisfy the condition provided with the filter. *Projection postfixes* compute a new tuple sequence based on every tuple in a sequence.

EXAMPLE:
The following path expression computes a table with two columns:

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

As starting point it uses a map bound to the variable `%biblio`. The shortcut `// person` then filters out those items which are instances of `person`. This is followed by another postfix which takes each person as starting point (indicated by `.`) and computes the person's name for the first column and the person's authorship(s) for a second column.

The tuple expression or the simple content 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 the postfixes are applied in lexical order. The result of the last postfix application is the result of the path expression.

If the computed tuple sequence 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 according to the order in the incoming tuple sequence.

6.6.2 Filter Postfix

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

[52]	*filter-postfix*	::=	**[** *boolean-primitive* **]**

When the filter postfix is applied to an incoming tuple sequence, the boolean primitive will be evaluated *for every tuple* in this sequence. For this, each of these tuples will be bound one-by-one to a new instance of the variable `@_`. This binding is added to the context for the evaluation of the boolean primitive only.

Only tuples in the incoming tuple sequence where the evaluation of the boolean primitive returns a non-empty sequence will be included in the outgoing tuple sequence.

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

[T]	*filter-postfix*	::=	**//** *anchor*
		==>	**[** **^** *anchor* **]**

EXAMPLE:
To filter out all `person` instances from the map bound to `%map` one can use:

```
%map // person
```

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

[U]	*path-expression*	::=	**//** *anchor* { *postfix* }
		==>	**%** **//** *anchor* { *postfix* }

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:

[V]	*predicate-postfix*	::=	**[** *integer* **]**
		==>	**[** **$#** **==** *integer* **]**
[W]	*predicate-postfix*	::=	**[** *integer-1* **..** *integer-2* **]**
		==>	**[** *integer-1* **<=** **$#** **&** **$#** **<** *integer-2* **]**

EXAMPLE:
To select the third `person` from a tuple sequence, one can use:

```
// person [ 2 ]
```

EXAMPLE:
To select the first 10 `person`s from a tuple sequence, one can use:

```
// person [ 0 .. 10 ]
```

6.6.3 Projection Postfix

When operating on tuple sequences, it is sometimes necessary to select particular components out of the tuples in 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*:

[53]	*projection-postfix*	::=	*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:
The path expression

```
// opera ( . / name [ @ opus ], . <- work [ ^ is-composed-by ] )
```

extracts first all instances of `opera`s. Then a new tuple is generated for each 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 Predicate Invocations

A special form of path expressions are *predicate invocations*. With these a processor will look for association items of a certain type and with certain role/player configurations:

[54] *predicate-invocation* ::= *anchor* **(** < *anchor* **:** *value-expression* > [**,**] [**...**] **)**

The anchor before the opening bracket is interpreted to be an association type in the context map. Any number of role/player combinations can be specified. The role types are all items as specified with anchors. The players are computed via a value expression. The optional ellipsis (`...`) can be used to indicate that matching associations may contain other roles not constrained by the predicate invocation (*non-strict predicate invocation*). Otherwise, the predicate invocation is *strict*, in that no other roles are allowed.

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.

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

EXAMPLE:
Roles cannot be omitted. But the role can be `tm:subject` (shortcut `*`) if it does not matter:

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

In the current context, the anchors are evaluated. The results must be each a map item, otherwise an error is flagged.

In the current context all value expressions for the players are evaluated. The result of each such evaluation is a tuple sequence. All these sequences must contain singleton tuples with only topic items as values; otherwise an error is flagged. The sequence of items is regarded as *potential players* below.

The evaluation result of a predicate invocation is a singleton tuple sequence containing all association items in the context map which satisfy the following conditions:

1. The association item must be an instance of the given type, honoring subtype transitivity.
2. For each of the roles mentioned in the invocation, that association item must have a role which is a subtype (direct 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 roles in the association item which are not constrained by the predicate invocation.

NOTE:
While predicate invocations are a more concise notation for their purpose than general path expressions, they do not introduce new expressitivity.

Additional Notation: Following shorthand notations for *iko* (is subclass of) and for *isa* (is instance of) can be used:

[X] *path-expression* ::= *simple-content-1* **iko** *simple-content-2*
==> **tm:subclass-of** **(** **tm:subclass** **:** *simple-content-1* **,** **tm:superclass** **:** *simple-content-2* **)**

[Y] *path-expression* ::= *simple-content-1* **isa** *simple-content-2*
==> **tm:type-instance** **(** **tm:instance** **:** *simple-content-1* **,** **tm:type** **:** *simple-content-2* **)**

7 Formal Semantics

The initial binding set $B = \{ \%\% \Rightarrow e, v_1 \Rightarrow a_1, v_2 \Rightarrow a_2, ...\}$ contains a binding for the special variable `%%` to a map *e*, and an otherwise arbitrary set of additional variables $v_1, v_2, ...$ which are assigned to Topic Map items and other values. All values $a_1, a_2, ...$ are constants provided by the calling environment.

The formal semantics for TMQL describes the result of evaluating a query expression under this binding set:

$\text{eval}_{\text{TMDM}}$ (*query-expression* | *B*)

The vertical bar symbolizes that the evaluation must be the context of the binding set, i.e. that all free variables within the query expression which are in the list of bindings will be replaced by the value in that binding.

As primitive values we use those which are predefined by TMQL (integer, float, etc.), for topic maps this International Standard commits to those items following TMDM (Topic Maps data model). As TMDM is not directly amenable to a concise formalization a slightly different route is chosen:

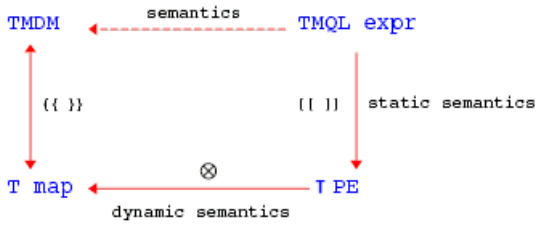

Figure 1 — Formal Semantics

The process assumes a structural mapping between items of a particular TMDM instance and proxies in a TMRM *subject map*. This mapping is symbolized with {{ }} and is formally defined in TMRM (Annex B). When applied to a TMDM item, the mapping will construct an equivalent proxy set. The mapping also includes constants (such as integers) and is reversible. {{ }} also induces another mapping of navigation steps on the TMDM level and the corresponding navigation on the TMRM level. For simplicity we continue to use {{ }} to symbolize this. $\{\{ \}\}^{-1}$ denotes the inverse mapping from TMRM to TMDM.

The *static semantics* for TMQL ⟦ ⟧ is then defined as a mapping of TMQL expressions onto TMRM path expressions (TMRM, Annex A): every valid syntactic form in the TMQL grammar is translated into these low level path expressions. The TMRM defines the result when a TMRM path expression is applied to a subject map (using the binary operator ⊗), so that implicitly the *dynamic semantics* $\text{eval}_{\text{TMDM}}$ of TMQL expressions applied to TMDM topic map is defined as:

$$\begin{aligned} &\text{eval}_{\text{TMDM}}\ (\ \textit{query-expression}\ |\ B\ ) \\ =\ &\{\{\ ⟦\ \textit{query-expression}\ |\ B\ ⟧\ \ \}\}^{-1} \end{aligned}$$

NOTE:
TMQL does not perform any inferencing on any map involved. This is all regarded to be outside the scope and a matter of the map access infrastructure.

7.1 Notation

The formal semantics deals with several layers:

1. TMQL syntax layer: On this layer non-terminals are set in *italic* and terminals are **bold and underlined**.
2. Grammar layer: Here the symbols {}, <>, [], () and | are used to express grammatical structures (3.1).
3. TMRM path expression level: Here the symbols are subscripted with τ.

To keep the notation succinct we adopt a number of shorthands, avoiding the use of indices as much as possible. So, for instance, we allow grammar symbols to move over ⟦⟧ pairs, such as in

$$\begin{aligned} ⟦\ <\ x\ >\ ⟧ &= ⟦\ x_1,\ x_2,\ \ldots,\ x_n\ ⟧ \\ &= ⟦\ x_1\ ⟧,\ ⟦\ x_2\ ⟧,\ \ldots,\ ⟦\ x_n\ ⟧ \\ &= <\ ⟦\ x\ ⟧\ > \end{aligned}$$

How such individual results $⟦\ x_i\ ⟧$ are combined, depends on the context. For a boolean expression it is the operator &, for lists the comma is used. For XML elements and TM content such repetitions are accumulated into a larger object, be it an XML fragment or a TM fragment. For this we also use ∑ to stress the summation aspect.

For variables we introduce also a shorthand to control the number of primes appended to the variable's name. v^4, for instance, is the same as v''''.

Occasionally we need to *exponentiate* the phrase matching a non-terminal, such as *boolean-expression*^{N}. For this process all free variables $\{ v_1, v_2, \ldots, v_m \}$ within the matched phrase have to be identified first. Then in the phrase all occurrences of the free variable v_1 is replaced by v_1^{m+1} where m is the biggest number of primes used within the phrase. Any occurrences of v_2 are replaced with v_2^{m+2}, and so forth. The same process is repeated with another copy of the phrase, this time starting with m+2 instead of m+1. At the end these phrases are combined, again leaving it up to the context how the individual results are combined.

7.2 Semantics of Variables

1. Variables differing only in the number of primes are regarded to never bind to the same value in one and the same binding set. To eliminate any primed variables v^m and v^n in *p* that semantic is simply made explicit:

   $$\begin{aligned} &⟦\ p\ |\ B\ ⟧ \\ =\ &⟦\ p\{[\ v^m \to u,\ v^n \to w\ ]\}\ \underline{\underline{[\ \textbf{not}\ u\ ==\ w\ ]}}\ |\ B\{[\ v^m \to u,\ v^n \to w\ ]\}\ ⟧ \end{aligned}$$

   where *u* and *v* are newly introduced variables (not occurring anywhere in *p*). Note that the substitution has to happen in *p* and also in all expressions in the variable assignments.
   The process is repeated until no more primed variable pairs exist in *p*. The order of this elimination is not relevant.
2. Any occurrences of $_ within an expression *p* is replaced with a new variable not occurring in *p*:

   $$\begin{aligned} &⟦\ p\ |\ B\ ⟧ \\ =\ &⟦\ p\{[\ \underline{\$\_}\ \to u\ ]\}\ |\ B\ ⟧ \end{aligned}$$

   Note that assignments are not affected.
3. Named variables can be completely eliminated from expressions, so that only positional variables $0, $1, ... remain in the down-translated expression. These positional variables directly correspond to projection:

   $$⟦\ \$i\ ⟧ = \pi_i$$

   The elimination of named variables depends on their sigil:

1. For variables with sigil `$` (they can only hold simple values), an expression *p* under a binding **$v =>** *q* will be translated as follows:

$$\begin{aligned}&[\![\ p\ |\ \$v\ \Rightarrow\ q\ ]\!]\\ =\ &[\![\ \text{fn:zigzag}\ (q)\ ]\!]\\ &\otimes_T\ [\![\ (\ \$0,\ p\{[\ \$v\ \rightarrow\ \$0\ ]\}\ )\ ]\!]\end{aligned}$$

   First the expression of the binding is iterated through (using the predefined `fn:zigzag` function). Then the original expression is extended by a leading `$0`. In the same step that expression is changed in that `$v` is replaced with `$0`, only after all instances of `$0` are replaced with `$1`, and that only after all instances of `$1` are replaced with `$2`, and so forth.
2. For variables with sigil `@` (they can hold single tuples, i.e. lists), an expression *p* under a binding **@v =>** *q* will be translated as follows:

$$\begin{aligned}&[\![\ p\ |\ @v\ \Rightarrow\ q\ ]\!]\\ =\ &[\![\ q\ ]\!]\\ &\otimes_T\ [\![\ (\ \$0,\ \$1,\ \ldots,\ \$n\text{-}1,\ p\{[\ @v\ \rightarrow\ (\$0,\ \$1,\ \ldots,\ \$n\text{-}1)\ ]\}\ )\ ]\!]\end{aligned}$$

   First the expression of the binding *q* is taken, then the original expression *p* is applied, but only after all occurrences of `@v` are replaced with the identity projection `$0`, `$1`, ..., *$n-1* with *n* the length of the tuples produced by *q*. All existing instances of *$i* variables are again shifted to higher indices *$i+n*.
3. For variables with the sigil `%` (they can hold tuple sequences), an expression *p* under a binding **%v =>** *q* will be translated as follows:

$$\begin{aligned}&[\![\ p\ |\ \%v\ \Rightarrow\ q\ ]\!]\\ =\ &[\![\ p\ \{[\ \%v\ \rightarrow\ q\ ]\}\ ]\!]\end{aligned}$$

If several variable assignments exist for an expression *p*, then the process is repeated for each of these variable assignments. The order of the elimination does not matter.

7.3 Semantics of Content

1. The static semantics of constants (4.1) is that provided by atoms and item references. The static semantics of atomic values (4.2) and item references (4.3) is that provided by the mapping to TMRM:

$$[\![\ atom\ ]\!] = \{\{\ atom\ \}\}$$

$$[\![\ item\text{-}reference\ ]\!] = \{\{\ item\text{-}reference\ \}\}$$

2. The static semantics of navigation steps (4.4) is also based on {{ }}:

$$[\![\ step\ ]\!] = \{\{\ (\ \gg\ |\ \ll\ )\ axis\ \ [\ [\![\ anchor\ ]\!]\ ]\ \}\}$$

3. The static semantics of simple content (4.6) is given by first mapping the anchor to TMRM and then applying the mapped navigation steps to it:

$$\begin{aligned}[\![\ simple\text{-}content\ ]\!] &= [\![\ anchor\ ]\!]\ \ [\ \otimes_T\ [\![\ navigation\ ]\!]\ ]\\ [\![\ navigation\ ]\!] &= [\![\ step\ ]\!]\ \ \ [\ \otimes_T\ [\![\ navigation\ ]\!]\ ]\end{aligned}$$

   The mapping $[\![\,]\!]$ guarantees that there are no variables left in any query subexpression. Therefore the anchor can only be a constant:

$$[\![\ anchor\ ]\!] = [\![\ constant\ ]\!]$$

4. The static semantics of composite content (4.7) created via binary operator is provided separately for each case:

$$\begin{aligned}[\![\ content_1\ \text{++}\ content_2\ ]\!] &= [\![\ content_1\ ]\!]\ +_T\ [\![\ content_2\ ]\!]\\ [\![\ content_1\ \text{--}\ content_2\ ]\!] &= [\![\ content_1\ ]\!]\ -_T\ [\![\ content_2\ ]\!]\\ [\![\ content_1\ ==\ content_2\ ]\!] &= [\![\ content_1\ ]\!]\ =_T\ [\![\ content_2\ ]\!]\end{aligned}$$

   If the content is generated by an unconditional nested query expression, then

$$[\![\ \{\ query\text{-}expression\ \}\ ]\!] = [\![\ query\text{-}expression\ ]\!]$$

   If it is created via a conditional, then (it is always guaranteed that an ELSE branch exists):

$$\begin{aligned}&[\![\ \text{if}\ path\text{-}expression\ \text{then}\ content_1\ \text{else}\ content_2\ ]\!]\\ =\ &[\![\ path\text{-}expression\ ]\!]\ ?_T\ [\![\ content_1\ ]\!]\ :_T\ [\![\ content_2\ ]\!]\end{aligned}$$

   Content can also be generated as TM fragment or as XML fragment. The static semantics is provided by $[\![\ tm\text{-}content\ ]\!]$ and $[\![\ xml\text{-}content\ ]\!]$, respectively.
5. The static semantics of a tuple expression (4.8.3) is straightforward if no ordering is involved:

$$\begin{aligned}&[\![\ (\ <\ value\text{-}expression\ >\ )\ ]\!]\\ =\ &(_T\ \ <\ [\![\ value\text{-}expression\ ]\!]\ >\ )_T\end{aligned}$$

   If ordering is requested, then the ordering tuple containing only `asc` and `desc` as values. It has the same arity as the length of the tuple expression, otherwise it will be padded with `asc`. For an ordered tuple expression the static semantics is then

$$\begin{aligned}&[\![\ (\ <\ value\text{-}expression_i\ order_i\ >\ )\ ]\!]\\ =\ &(_T\ \ <\ [\![\ value\text{-}expression_i\ ]\!]\ >\ )_T\ \otimes_T\ \text{sort}_T(\ldots,\ order_i,\ \ldots)\end{aligned}$$

   Every `asc` in the ordering tuple is translated into ↑ and every instance of `down` into ↓.

6. The static semantics for XML content (4.9) is provided by propagating ⟦ ⟧ down the XML fragment. The top-level elements are then treated as atomic values and are concatenated into a tuple sequence. Individual XML elements are collected and then are concatenated into an XML fragment.

$$⟦ \{ \textit{xml-element} \} ⟧ = \sum \{ ⟦ \textit{xml-element} ⟧ \}$$

The sum is built with the predefined operator $++_{XML}$ defined on XML. Individual XML element nodes are built with an intrinsic function $Element_{DOM}$:

$$\begin{aligned} &⟦ \textbf{<} \textit{ xml-id } \{ \textit{ xml-attribute } \} \textit{ xml-rest } ⟧ \\ &= Element_{DOM}\ (\text{tagName} \quad : ⟦ \textit{xml-id} ⟧, \\ &\qquad\qquad \text{attributes} : \{ ⟦ \textit{xml-attribute} ⟧ \}\ , \\ &\qquad\qquad \text{childNodes} : ⟦ \textit{xml-rest} ⟧) \end{aligned}$$

and the element content is recursively built with:

$$\begin{aligned} &⟦ \textbf{>} \{ \textit{xml-element} \mid \textit{xml-fragment} \} ⟧ \\ &= \sum \{ ⟦ \textit{xml-element} \mid \textit{xml-fragment} ⟧ \} \end{aligned}$$

XML fragments are then either further XML elements or text nodes. These text nodes may contain nested query expressions, so that ⟦ ⟧ has to be propagated as well.

7. The static semantics for TM content (4.10) is provided by first propagating ⟦ ⟧ into the CTM instance. From that instance then a TMDM instance fragment is generated via deserialization. That, in turn, is mapped via {{}} to an TMRM subject map fragment:

$$⟦ \textit{tm-content} ⟧ = \{\{ \text{deserialize}_{TMDM} (⟦ \textit{ctm-instance} ⟧) \}\}$$

Everywhere where a nested query expression is used within the CTM instance, that has to be mapped to ⟦ *query-expression* ⟧. Since the CTM syntax is delegated, this is not further detailed here.

8. The static semantics of a function invocation (4.12) can only be given to the extent to which functions are known to a TMQL processor. Every function is manifested as topic in the environment map, but any information about a function's implementation is outside the scope of this International Standard.
For a function invocation with positional parameter association, the static semantics is

$$\begin{aligned} &⟦ \textit{item-reference tuple-expression} ⟧ \\ &= \{\{ \textit{item-reference} \}\}\ (_T\ ⟦ \textit{tuple-expression} ⟧\ )_T \end{aligned}$$

For the named parameter association it is:

$$\begin{aligned} &⟦ \textit{item-reference } \textbf{(}\ \textit{identifier}_i\ \textbf{:}\ \textit{value-expression}_i\ \textbf{)} ⟧ \\ &= \{\{ \textit{item-reference} \}\}\ (_T\ ⟦ \textit{value-expression}_i ⟧\ )_T \end{aligned}$$

9. The static semantics of boolean expressions (4.13) are provided by the following:

$$\begin{aligned} &⟦ \textit{boolean-expression}_1 \textbf{ | } \textit{boolean-expression}_2 ⟧ \\ &= (_T\ ⟦ \textit{boolean-expression}_1 ⟧\ +_T\ ⟦ \textit{boolean-expression}_2 ⟧\ )_T \end{aligned}$$

$$\begin{aligned} &⟦ \textit{boolean-expression}_1 \textbf{ \& } \textit{boolean-expression}_2 ⟧ \\ &= (_T\ ⟦ \textit{boolean-expression}_1 ⟧\ \otimes_T\ ⟦ \textit{boolean-expression}_2 ⟧\ )_T \end{aligned}$$

Boolean primitives either nest properly:

$$⟦ \textbf{(}\ \textit{boolean-expression}\ \textbf{)} ⟧ = (_T\ ⟦ \textit{boolean-expression} ⟧\ )_T$$

or they negate:

$$\begin{aligned} &⟦ \textbf{not } \textit{boolean-primitive} ⟧ \\ &= ⟦ \textit{boolean-primitive} ⟧\ ?_T\ ⟦\ 0_T ⟧\ :_T\ ⟦\ (1)\ ⟧ \end{aligned}$$

The static semantics of numerically unrestricted EXISTs clauses (4.13.2) is

$$\begin{aligned} &⟦ \textbf{some } \textit{binding-set } \textbf{satisfies } \textit{boolean-expression} ⟧ \\ &= ⟦ \textit{boolean-expression} ⟧ \mid ⟦ \textit{binding-set} ⟧ \end{aligned}$$

If the EXISTs clause has a lower bound *N*, then the semantics amounts to

$$\begin{aligned} &⟦ \textbf{at least } \textit{integer binding-set } \textbf{satisfies } \textit{boolean-expression} ⟧ \\ &= ⟦ \textit{boolean-expression}^{integer} ⟧ \mid ⟦ \textit{binding-set}^{integer} ⟧ \end{aligned}$$

If the EXISTs clause uses an upper bound then the static semantics is given by

$$\begin{aligned} &⟦ \textbf{at most } \textit{integer binding-set } \textbf{satisfies } \textit{boolean-expression} ⟧ \\ &= ⟦ \textbf{not } \textit{boolean-expression}^{integer+1} ⟧ \mid ⟦ \textit{binding-set}^{integer+1} ⟧ \end{aligned}$$

7.4 Semantics of Query Contexts

1. The static semantics of a variable assignment (5.2) is a pair consisting of the variable itself and the semantics of the content expression associated with it:

$$⟦ \textit{variable } \textbf{in } \textit{content} ⟧ \quad = \textit{variable} \Rightarrow ⟦ \textit{content} ⟧$$

2. If several variable assignments are used to create a binding set, then these pairs are combined:

$$⟦ < \textit{variable-assignment} > ⟧ = < ⟦ \textit{variable-assignment} ⟧ >$$

3. If the binding sets have to be sorted, then this is done with an ordering value expression:

$$< \textit{variable-assignment} >$$

```
sorted-by < value-expression [ asc | desc ] >
```

For this purpose the value expression is evaluated with each binding set

```
0 = < value-expression > [ 0 ]
```

selecting only the first tuple in each of the intermediate results. The result tuple sequence *s* of the expression *O* | ⟦ < *variable-assignment* > ⟧ is then ordered according to the order direction tuple o = ⟦ ..., **asc**, ..., **desc**, ... ⟧ whereby

```
⟦ asc ⟧  = ↑
⟦ desc ⟧ = ↓
```

The individual binding sets are then ordered according to the order produced by *s* sort_T^o so that a binding set *B* is before another *B*' if the tuple produced with *O* | *B* comes before that of *O* | *B*'.

7.5 Semantics of Query Expressions

1. The meaning of a SELECT expressions (6.4) is controlled via several clauses. The intention of the FROM clause is to redefine the context map:

```
⟦ select < value-expression1 >
  [ from value-expression2 ]
  [ where boolean-expression ]
  [ order by < value-expression3 > ]
  [ unique ]
  [ offset value-expression4 ]
  [ limit value-expression5 ] ⟧
= ⟦ for %_ in value-expression2
  return {
      select < value-expression1 >
        from %
      [ where boolean-expression ]
      [ order by < value-expression3 > ]
      [ unique ]
      [ offset value-expression4 ]
      [ limit value-expression5 ]
  } ⟧
```

The OFFSET, LIMIT and UNIQUE clauses only operate on the result tuple sequence of the SELECT clause. The UNIQUE clause removes the duplicate tuples:

```
⟦ select < value-expression1 >
    from %_ ]
  [ where boolean-expression ]
  [ order by < value-expression3 > ]
    unique
  [ offset value-expression4 ]
  [ limit value-expression5 ] ⟧
= ⟦ select < value-expression1 >
    from %_ ]
  [ where boolean-expression ]
  [ order by < value-expression3 > ]
  [ offset value-expression4 ]
  [ limit value-expression5 ] ⟧
  ⊗T uniqT
```

and OFFSET and LIMIT slice off a subsequence starting at the offset value:

```
⟦ select < value-expression1 >
    from %_ ]
  [ where boolean-expression ]
  [ order by < value-expression3 > ]
    offset value-expression4
    limit value-expression5 ⟧
= ⟦ select < value-expression1 >
    from %_ ]
  [ where boolean-expression ]
  [ order by < value-expression3 > ]
  [T ⟦ value-expression4 ⟧ .. ⟦ value-expression4 ⟧ +INT ⟦ value-expression5 ⟧ ]T
```

With these out of the way, the remaining static semantics simplifies to:

```
⟦ select < value-expression1 >
    from %_ ]
  [ where boolean-expression ]
  [ order by < value-expression2 [ asc | desc ] > ] ⟧
= (T ⟦ boolean-expression ⟧ ?T 1T :T ⟦ 0T ⟧ )T
  ⊗T ⟦ < value-expression1 > ⟧
  | v1 ⇒ ⟦ %_ ⟧, ..., vn ⇒ ⟦ %_ ⟧
    sorted-by < value-expression2 [ asc | desc ] >
```

whereby the v_1, ..., v_n are the *free* variables within the *boolean-expression*.

2. The static semantics of a FLWR expression (6.5) is provided by computing the content of the RETURN clause after having generated variable binding sets which are filtered by the boolean expression in the WHERE clause:

```
 ⟦ [ for binding-set ]
   [ where boolean-expression ]
   [ order by < value-expression [ asc | desc ] > ]
     return content ⟧
=
  (T ⟦ boolean-expression ⟧ ?T 1T :T ⟦ 0T ⟧ )T
```

```
⊗T ⟦ content ⟧
| ⟦ [ binding-set ] ⟧
  sorted-by < value-expression [ asc | desc ] >
```

3. A path expression (6.6) can either be a genuine path expression with postfixes or a predicate invocation. A genuine path expression has its static semantics defined via the application of postfixes:

```
  ⟦ ( tuple-expression | simple-content ) { postfix } ⟧
= ⟦ ( tuple-expression | simple-content ) ⟧ ⊗T { ⟦ postfix ⟧ }
```

Postfixes themselves are either filters or projections:

```
  ⟦ [ boolean-primitive ] ⟧
= ⟦ boolean-primitive ⟧ ?T 1T :T ⟦ 0T ⟧


  ⟦ projection-postfix ⟧
= ⟦ tuple-expression ⟧
```

The predicate invocation (6.6.4) is a only specialization of a genuine path expression and can be expressed in terms of it. In the non-strict case, the static semantics is:

```
  ⟦ anchor ( < anchori : value-expressioni > , ... )
= ⟦ // anchor < [ . -> anchori == value-expressioni ] > ⟧
```

Hereby first all associations of the required type are extracted and then filtered whether they also contain a role with the specified role types and appropriate values.

The strict case uses the above definition and additionally filters out those associations which have additional role types not mentioned:

```
  ⟦ anchor ( < anchori : value-expressioni > )
= ⟦ ( anchor ( < anchori : value-expressioni > , ... ) )
    [ not . >> roles -- ∑ anchori ] ⟧
```

8 Conformance

A processor is conformant with this specification if:

1. It accepts as valid query expressions every character stream following the syntactical und semantic constraints as defined and rejects all other streams with flagging errors at those situations described.
2. It provides a minimal environment as defined in Annex A.

   NOTE:
   Processors may provide a richer environment. Specifically, they may procure additional data types in which case they have to define equality, total ordering between values and optionally also operators for this data type. They also have to specify stringification rules to convert strings in the lexical value space to internal values and back.
3. It delivers all evaluation results for valid query expressions according to the (formal) semantics and it reports error conditions for all conditions defined here.

   NOTE:
   Processor may deliver results eagerly or lazily.

A Predefined Environment (normative)

```
tmql   http://psi.isotopicmaps.org/tmql/1.0
-: TMQL
-: "Topic Maps Query Language" @long
description: textual language to extract content from TM-based backends
version    : 1.0

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

tmql-concept
-: TMQL core concept

ontology isa tmql-concept
-: Ontology

function isa tmql-concept
-: Function

predicate iko function
-: Predicate

binary-operator  iko function
-: Binary Operator

unary-operator   iko function
-: Unary Operator

prefix-operator  iko unary-operator
-: Prefix (unary) operator

postfix-operator iko unary-operator
-: Postfix (unary) operator

#-- data types ---------------------------------------------------

datatype isa tmql-concept
-: Data Type

primitive-datatype iko datatype
-: Primitive Data Type

undefined isa primitive-datatype
-: Undefined Datatype
description: has only a single value

boolean  isa primitive-datatype http://www.w3.org/TR/xmlschema-2/datatypes.html#boolean
```

```
-: Boolean

integer  isa primitive-datatype http://www.w3.org/TR/xmlschema-2/datatypes.html#integer
-: Integer Number

decimal  isa primitive-datatype http://www.w3.org/TR/xmlschema-2/datatypes.html#decimal
-: Decimal Number

iri      isa primitive-datatype http://www.w3.org/TR/xmlschema-2/datatypes.html#anyURI
-: IRI (RFC 3987)

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

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

string   isa primitive-datatype http://www.w3.org/TR/xmlschema-2/datatypes.html#string
-: String

xml      isa primitive-datatype
-: XML Content

complex-datatype iko datatype
-: Complex Data Type

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

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

#-- functions and operators ------------------------------------------------

#-- for decimals

decimal-unary-add isa prefix-operator
-: "+" @ prefix-notation
profile: fn:add-decimal (a : decimal) return decimal

decimal-unary-minus isa prefix-operator
-: "-" @ prefix-notation
profile: fn:minus-decimal (a : decimal) return decimal

decimal-binary-add isa binary-operator
-: "+" @ infix-notation
profile: fn:add-decimal (a : decimal, b : decimal) return decimal

decimal-binary-minus isa binary-operator
-: "-" @ infix-notation
profile: fn:minus-decimal (a : decimal, b : decimal) return decimal

decimal-binary-mul isa binary-operator
-: "*" @ infix-notation
profile: fn:mul-decimal (a : decimal, b : decimal) return decimal

decimal-binary-div isa binary-operator
-: "%" @ infix-notation
profile: fn:div-decimal (a : decimal, b : decimal) return decimal

decimal-binary-mod isa binary-operator
-: "mod" @ infix-notation
profile: fn:mod-decimal (a : decimal, b : decimal) return decimal

decimal-binary-lt isa binary-operator
-: "<" @ infix-notation
profile: fn:lt-decimal (a : decimal, b : decimal) return tuple-sequence

decimal-binary-le isa binary-operator
-: "<=" @ infix-notation
profile: fn:leq-decimal (a : decimal, b : decimal) return tuple-sequence

decimal-binary-gt isa binary-operator
-: ">" @ infix-notation
profile: fn:gt-decimal (a : decimal, b : decimal) return tuple-sequence

decimal-binary-ge isa binary-operator
-: ">=" @ infix-notation
profile: fn:geq-decimal (a : decimal, b : decimal) return tuple-sequence

#-- for strings

string-concat isa binary-operator
-: "+" @ infix-notation
profile: fn:concat (a : string, b : string) return string

string-length isa function
profile: fn:length (s : string) return integer

string-less-than isa binary-operator
-: "<" @ infix-notation
profile: fn:string-lt (a : string, b : string) return tuple-sequence

string-less-equal-than isa binary-operator
-: "<=" @ infix-notation
profile: fn:string-leq (a : string, b : string) return tuple-sequence

string-greater-equal-than isa binary-operator
-: ">=" @ infix-notation
profile: fn:string-geq (a : string, b : string) return tuple-sequence

string-greater-than isa binary-operator
-: ">" @ infix-notation
profile: fn:string-gt (a : string, b : string) return tuple-sequence

string-regexp-match isa binary-operator
-: "=~" @ infix-notation
```

```
profile: fn:regexp (s : string, re : string) return tuple-sequence

#-- for tuple sequences

slicing  isa function
profile: fn:slice (s : tuple-sequence, low : integer, high : integer) return tuple-sequence
description: selects those tuples with index between low and high-1

count    isa function
profile: fn:count (s : tuple-sequence) return integer
description: returns the number of tuples in the sequence

uniq     isa function
profile: fn:uniq (s : tuple-sequence) return tuple-sequence
description: returns a new tuple sequence with all duplicate tuples suppressed

plusplus isa binary-operator
-: "++" @ infix-notation
profile: fn:concat (s : tuple-sequence, t : tuple-sequence) return tuple-sequence
description: produces a tuple sequence with all tuples combined -- any ordering is honored

minusminus isa binary-operator
-: "--" @ infix-notation
profile: fn:except (s : tuple-sequence, t : tuple-sequence) return tuple-sequence
description: produces a tuple sequence where all tuple which appear in t are removed from s

eqeq isa binary-operator
-: "==" @ infix-notation
profile: fn:compare (s : tuple-sequence, t : tuple-sequence) return tuple-sequence
description: produces a tuple sequence of all tuples which appear in s and t

zigzag isa function
profile: fn:zigzag (s : tuple-sequence) return tuple-sequence
description: """
returns a singleton sequence filled with all values from all tuples
index of tuples run faster than index within the tuple sequence
"""

zagzig isa function
profile: fn:zagzig (s : tuple-sequence) return tuple-sequence
description: """
returns a singleton sequence filled with all values from all tuples
index within the tuple sequence run faster than index within one tuple
"""

concat isa function
profile: fn:concat (s : tuple-sequence) return tuple-sequence
description: """
returns a single tuple which is the concatenation of all tuples in the sequence
"""
```

B Delimiting Symbols (normative)

Following characters are delimiting:

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

C Syntax (informative)

Core Syntax

[12]	*IRI*	::=	**([^<>'{}|^`] - [#x00-#x20])***
[11]	*QIRI*	::=	*IRI* | *QName*
[13]	*QName*	::=	*prefix* *identifier*
[19]	*anchor*	::=	*constant* | *variable*
[2]	*atom*	::=	*undefined* | *boolean* | *integer* | *decimal* | *iri* | *date* | *dateTime* | *string* [**^^** *QIRI*]
[18]	*axis*	::=	**types** | **supertypes** | **players** | **roles** | **characteristics** | **scope** | **locators** | **indicators** | **reifier** | **atomify**
[44]	*binding-set*	::=	< *variable-assignment* >
[4]	*boolean*	::=	**true** | **false**
[37]	*boolean-expression*	::=	*boolean-expression* **|** *boolean-expression* | *boolean-expression* **&** *boolean-expression* | *boolean-primitive*
[R]	*boolean-expression*	::=	**every** *binding-set* **satisfies** *boolean-expression*
		==>	**not** **some** *binding-set* **satisfies** **not** **(** *boolean-expression* **)**

```
[38]  boolean-primitive     ::=  ( boolean-expression ) |
                                 not boolean-primitive  |
                                 forall-clause          |
                                 exists-clause
[N]   boolean-primitive     ::=  ^ anchor
                            ==>  . >> types == anchor
[O]   boolean-primitive     ::=  @ anchor
                            ==>  . @ == anchor
[1]   constant              ::=  atom | item-reference
[22]  content               ::=  content ( ++ | -- | == ) content          |
                                 { query-expression }                     |
                                 if path-expression then content [ else content ] |
                                 tm-content                               |
                                 xml-content
[K]   content               ::=  path_expression
                            ==>  { path_expression }
[L]   content               ::=  path-expression-1 || path-expression-2
                            ==>  if path-expression-1 then { path-expression-1 } else {
                                 path-expression-2 }
[7]   date                  ::=  ...http://www.w3.org/2001/XMLSchema#date...
[8]   dateTime              ::=  ...http://www.w3.org/2001/XMLSchema#dateTime...
[6]   decimal               ::=  /[+-]?\d+(\.\d+)?/
[46]  environment-clause    ::=  """ ... """
[39]  exists-clause         ::=  exists-quantifier binding-set satisfies boolean-expression
[P]   exists-clause         ::=  exists content
                            ==>  some $ in content satisfies not null
[Q]   exists-clause         ::=  content
                            ==>  exists content
[40]  exists-quantifier     ::=  some | at least integer | at most integer
[52]  filter-postfix        ::=  [ boolean-primitive ]
[T]   filter-postfix        ::=  // anchor
                            ==>  [ ^ anchor ]
[48]  flwr-expression       ::=  [ for binding-set ]
                                 [ where boolean-expression ]
                                 [ order by < value-expression > ]
                                 return content
[41]  forall-clause         ::=  every binding-set satisfies boolean-expression
[35]  function-invocation   ::=  item-reference parameters
[15]  identifier            ::=  /\w[\w\-\.]*/
[33]  infix-operator        ::=  ...any in the predefined environment...
[5]   integer               ::=  /[+-]?\d+/
[9]   iri                   ::=  < QIRI > | QIRI
[16]  item-reference        ::=  identifier | QIRI
[A]   item-reference        ::=  *
                            ==>  tm:subject
[I]   navigation            ::=  / anchor [ navigation ]
                            ==>  >> characteristics anchor >> atomify [ navigation ]
[J]   navigation            ::=  \ anchor [ navigation ]
                            ==>  << atomify << characteristics anchor [ navigation ]
[21]  navigation            ::=  step [ navigation ]
[36]  parameters            ::=  tuple-expression                         |
                                 ( < identifier : value-expression > )
[49]  path-expression       ::=  postfixed-expression | predicate-invocation
[U]   path-expression       ::=  // anchor { postfix }
                            ==>  % // anchor { postfix }
[X]   path-expression       ::=  simple-content-1 iko simple-content-2
                            ==>  tm:subclass-of ( tm:subclass : simple-content-1 ,
                                 tm:superclass : simple-content-2 )
[Y]   path-expression       ::=  simple-content-1 isa simple-content-2
                            ==>  tm:type-instance ( tm:instance : simple-content-1 , tm:type
                                 : simple-content-2 )
[51]  postfix               ::=  filter-postfix | projection-postfix
[50]  postfixed-expression  ::=  ( tuple-expression | simple-content ) { postfix }
[54]  predicate-invocation  ::=  anchor ( < anchor : value-expression > [ , ] [ ... ] )
[V]   predicate-postfix     ::=  [ integer ]
                            ==>  [ $# == integer ]
[W]   predicate-postfix     ::=  [ integer-1 .. integer-2 ]
                            ==>  [ integer-1 <= $# & $# < integer-2 ]
[14]  prefix                ::=  /\w+:/
[34]  prefix-operator       ::=  ...any in the predefined environment...
[53]  projection-postfix    ::=  tuple-expression
[45]  query-expression      ::=  [ environment-clause ]
                                 ( select-expression | flwr-expression | path-expression )
```

```
[47]  select-expression   ::=  select < value-expression >
                               [ from value-expression ]
                               [ where boolean-expression ]
                               [ order by < value-expression > ]
                               [ unique ]
                               [ offset value-expression ]
                               [ limit value-expression ]
[20]  simple-content      ::=  anchor [ navigation ]
[17]  step                ::=  ( >> | << ) axis [ anchor ]
[B]   step                ::=  >> instances
                          ==>  << types
[C]   step                ::=  -> anchor
                          ==>  >> players anchor
[D]   step                ::=  <- anchor
                          ==>  << players anchor
[E]   step                ::=  @
                          ==>  >> scope
[F]   step                ::=  =
                          ==>  << locators
[G]   step                ::=  ~
                          ==>  << indicators
[H]   step                ::=  ~~>
                          ==>  >> reifier
[10]  string              ::=  /"([^"]|\")*"/  |  /'([^']|\')*'/
[31]  tm-content          ::=  """ ctm-instance """
[23]  tuple-expression    ::=  ( < value-expression [ asc | desc ] > )
[M]   tuple-expression    ::=  null
                          ==>  ( )
[3]   undefined           ::=  undef
[32]  value-expression    ::=  value-expression infix-operator value-expression  |
                               prefix-operator value-expression                  |
                               function-invocation                               |
                               content
[42]  variable            ::=  /[$@%][\w#]+'*/
[S]   variable            ::=  .
                          ==>  $0
[43]  variable-assignment ::=  variable in content
[27]  xml-attribute       ::=  [ prefix ] xml-fragments = " xml-fragments "
[24]  xml-content         ::=  { xml-element }
[25]  xml-element         ::=  < xml-tag { xml-attribute } xml-rest
[29]  xml-fragments       ::=  { xml-text | { query-expression } }
[28]  xml-rest            ::=  > { xml-element | xml-fragments } </ xml-tag > |
                               />
[26]  xml-tag             ::=  [ prefix ] xml-fragments
[30]  xml-text            ::=  ...see text...
```

Bibliography

TMQLreq, *TMQL Requirements*, ISO, 2003, http://www1.y12.doe.gov/capabilities/sgml/sc34/document/0448.htm

TMQLuc, *TMQL Use Cases*, ISO, 2003, http://www.isotopicmaps.org/tmql/use-cases.html