Topic Maps Query Language

2007−01−22

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 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, @@@@@@@@@@@@

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

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

RFC3986, *RFC 3986 - Uniform Resource Identifiers (URI): Generic Syntax* , 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

3.1 Syntax Conventions

The syntax is defined on three levels:

1. At the *token (terminal) level* this International Standard makes use of regular expressions [RegExp] to specify character patterns for valid terminal symbols. They are enclosed in `//` to contrast them to constant terminals. 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.

   Terminals not defined in the grammar are (1) those for binary infix and unary prefix operators; these are specified in the predefined environment together with their function counterparts (Clause 7). And (2) those for `xsd:dateTime` and `anyURI` which refer to [XSDT].

   As usual, character symbols are either *delimiting* or not. The list of delimiting symbols is provided in Annex A. Any other symbol is not delimiting and whitespace characters (blank, tab and newlines) must be used for terminal separation. Whitespace characters are allowed everywhere between two terminals; 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 list, we use `< A >`. Productions are numbered for reference.
3. On top of the canonical syntax, a *non-canonical syntax level* introduced to reduce the syntactic noise in actual query expressions using shortcuts. 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 and its expanded form are semantically equivalent, so shortcuts do not add any computational complexity to the language. These mappings are numbered with letters for reference.

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

Annex B 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.

Ed. Note:
Every section which contains @@@@@@'s is still in limbo. Everything else claims not to be.

Ed. Note:
For a compact TM notation we use here AsTMa= 2.0 AsTMa[4] until CTM has emerged.

3.2 Ontological Commitments

TMQL is ontologically neutral, with the exception of *class transitivity* and *class membership*:

1. For the binary relation between a subclass and a superclass the interpretation and representation in [TMDM] (7.3) is adopted. Accordingly, for *transitivity* TMQL assumes that if a concept B is a superclass of A, and C is a superclass of B, then also C is a superclass of A. TMQL also interprets such relation as *reflexive* so that every class is a subclass and a superclass of itself.
2. For the binary relation between an instance and a class the interpretation and representation in [TMDM] (7.2) is adopted. Accordingly, TMQL assumes that if a concept is an instance of a class C, that very concept is also an instance of all superclasses of C.

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

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

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.

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

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

3.3 Informal and Formal Semantics

> The semantics of TMQL is defined in prose and also formally. The prose semantics is highlighted in the text using a vertical sidebar (as 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. The *static semantics* (Clause 9 maps every canonical TMQL expression onto TMRM path expressions ([TMRM], Appendix A). The *dynamic semantics* is given by the evaluation function for TMRM path expressions which operate on TMRM instance data. The semantics for TMDM instances follows indirectly via the mapping of TMDM instances into TMRM instances ([TMRM], Appendix B).

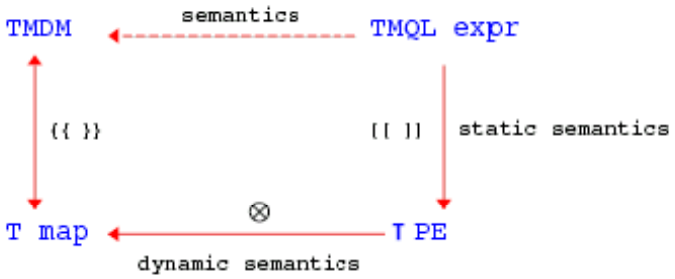

Figure 1 — Formal Semantics

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, dates or integers, and IRIs. This International Standard recognizes natively a small set of primitive data types (seeClause 7) together with operators on objects of these types.

```
[2]  atom            ::=  boolean                    |
                          integer                    |
                          decimal                    |
                          iri                        |
                          date                       |
                          string  [  ^^  iri-or-qname ]
[3]  boolean         ::=  true  |  false
[4]  integer         ::=  /-?\d+/
[5]  decimal         ::=  /-?\d+(\.\d+)?/
[6]  date            ::=  ...xsd:dateTime...
[7]  iri             ::=  "  iri-or-qname  "
[8]  iri-or-qname    ::=  ...xsd:anyURI...  | qname
[9]  qname           ::=  prefix identifier
[10] prefix          ::=  /\w+:/
[11] identifier      ::=  /\w[\w\-\.]*/
[12] string          ::=  /"[^"]*"/  |  /'[^']*'/
```

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

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

```
"23"^^xsd:integer
23                                  (xsd:integer)
3.1415                              (xsd:decimal)
2005-10-16T10:29                    (xsd:date)
"Hello World"                       (xsd:string)
"http://example.org/something"      (xsd:anyURI)
"http://example.org/something"^^xsd::string  (xsd:string)
```

```
"#@**()($^%$@$#@"^^what:ever              (what:ever)
```

The following are invalid atoms:

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

If a prefix is used to form a QName, then no blanks between the prefix and the following identifiers are allowed.

4.3 Item References

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

[13] *item-reference* ::= *identifier* | *qname* | *iri*

1. If the item reference is an *identifier* then this identifier is interpreted as an *item identifier* ([TMDM], 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 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). 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 (Clause 7). 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.

3. If the item reference is an *absolute IRI*, then that is used as subject indicator for a topic in the effective map. An error will be flagged if no such topic exists.

   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 everything is an instance of *subject* by definition, this makes predicates such as `is-located-in (* : $place, location: paris)` more robust. It also may impact performance, though, as no index can be used.

4.4 Navigation

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

```
[14] step      ::= ( >> | << ) axis [ item-reference ]
[15] axis      ::= classes      |
                   superclasses |
                   players      |
                   roles        |
                   characteristics |
                   scope        |
                   locators     |
                   indicators   |
                   reifier      |
                   atomify
```

The optional item reference adds typing information which is useful with some axes. If it is missing, tm:subject will be assumed.

Whenever this typing identifier is used, then class transitivity is honored, i.e. not only the identified item is considered but also all its subclasses.

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

classes

In forward direction this step computes all *classes* (types) of the value according to 3.2. In backward direction this step produces all *instances* of the value. The optional item identifier has no relevance.

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

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

```
[B] step       ::=  >>  instances
               ==>  <<  classes
```

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

superclasses

In forward direction this step computes all *superclasses* of the value according to 3.2. In backward direction this step produces all *subclasses* of the value. The optional item identifier has no relevance.

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

Additional Notation: Asking for all subclasses of an item is the inverse of asking for superclasses.

```
[C] step       ::=  >>  subclasses
               ==>  <<  superclasses
```

players

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

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

EXAMPLE:
The following navigation finds first all associations where a topic item (bound to $p) is

playing the role member. Then for all 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:

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

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

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

roles

If the value is an association item, in forward direction this step computes all *role-typing* topics. Multiple uses of the same role in one association causes multiple results. The optional item identifier has no relevance.

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

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

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

```
jack >> characteristics homepage
```

EXAMPLE:
To retrieve all characteristics one can use jack >> characteristics, for all names one can use jack >> characteristics name, for all occurrences jack >> characteristics occurrence.

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

```
$t >> characteristics >> classes
```

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 reference has no relevance.

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

```
[F] 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 identifier has no relevance.

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

```
[G] 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 would exist 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 identifier has no relevance.

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

```
[H] 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 or characteristics item which is reified by this topic. If the topic reifies a map, then all items in the map will be returned and the context map %_ will be set to this for the remainder of the directly enclosing TMQL expression. The optional item reference has no relevance.

If the value is an association or a characteristic 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:

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

If the value is an atom, in backward direction this step *de-atomifies* immediately the atom and returns all characteristics where this atom is used as data value. Also here the optional item identifier 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 characteristic with a certain URL:

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

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

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

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

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

```
23 \ age
```

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

All combinations of not listed above will render empty results. 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 characteristics are experienced in an ambivalent way: either as characteristics item, including not only the data value, but also the scope and the type of the characteristic; or as the atomic value alone.

When a characteristic item is subjected to an *atomify* navigation step, the atomic value is not immediately

extracted, but the process has to be postponed. If the atomification would be done immediately, the information for scope (and type) would be lost for further processing steps.

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

```
select $p / homepage ( @ wikipedia )
where
  $p isa person
```

Instead, atomification is postponed until one of the following situations occur:

1. The characteristics is about to be passed to the environment as part of the result.

   EXAMPLE:
   The following query expression will return the homepage URLs as IRIs (and not occurrence items):

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

2. The characteristics is about to be compared to an atomic value.

   EXAMPLE:
   The following query expression will return all persons younger than 42:

   ```
   select $p
   where
     $p / age < 42
   ```

3. The characteristics is about to be compared in the process of ordering (asc or desc, 6.5, 6.4, 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
   ```

4. The characteristics 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
   ```

5. The characteristics 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 $p
   where
     is-younger ($p, methusalem / age) &
     $p isa person
   ```

6. The characteristics is used inside an XML fragment.

   EXAMPLE:
   In the following, the name characteristics of each book will be atomified so that it can be inserted as text into the XML stream:

   ```
   return
   ```

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

7. The characteristics is used inside a TM fragment, except when used on positions where a characteristic declaration is expected ([CTM], Characteristics Declarations, @@@@@).

   EXAMPLE:
   The following query expression will return a TM fragment with persons from the context map 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 $p in // person
where
  is-younger ($p, methusalem / age)
return """

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

"""
```

8. The characteristics is subjected to a *de-atomification* step.

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

```
$book / name ( @ english ) \ title
```

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

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

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

4.6 Simple Content

Simple content is either an atomic value which refers to a constant, a reference to an information item in the context map, or a variable, any of which followed by an arbitrary number of navigation steps:

[16] *simple-content*	::=	*anchor* [*navigation*]
[17] *anchor*	::=	*constant* | *variable*
[18] *navigation*	::=	*step* [*navigation*]

First the anchor is evaluated in the current context. In any case the result is a tuple sequence. That 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.

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. Examples of atomic values are integers and strings, but can also be Topic Map items and XML fragments.

Content can be constructed in different ways:

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

When the binary infix operators ++, -- or == are used, -- and ++ are interpreted from left-to-right (left-associative). The operator == has the highest precedence.

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.

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 (. >> classes == person) inside the predicate each item from the %map is checked whether one of its classes is person.

        The map, a tuple sequence itself, will be iterated through. Inside the filter the first component of every individual tuple is addressed via . (which shortcuts $0). For this the list of classes (immediate and transitive ones) is computed. That tuple sequence is then compared with a trivial one containing only one singleton tuple with the 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*, concatenation).
    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 {}.
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:

[L]	*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:

[M]	*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 string `"undefined"` will be used.

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

4.8 Tuples and Tuple Sequences

4.8.1 Tuples

Tuples are ordered collections of simple values (atoms and items) whereby the values may be of different type (heterogeneous tuples).

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

The length of the tuple is called the *arity of the tuple*. As individual values of a tuple are ordered, the first value is assigned the index `0`, the next `1`, etc. *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 arity of a tuple, the extraction will result in an empty tuple sequence (4.8.3).

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

EXAMPLE:
The following query expression assigns first iteratively `name`/`homepage` pairs to a variable `@a`. In the RETURN clause then the two columns are swapped:

```
for @a in // person ( . / name, . / homepage )
return
   @a ($1, $0)
```

4.8.2 Comparing Tuples

> Tuples are only then equivalent, if they have the same length and all their individual values 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 by starting with index `0` moving by one to the next higher index at a draw.

   If the order tuple has `asc` as value on a given index, that tuple with the smaller component on that index is also the smaller tuple. If the ordering tuple has `desc` as value on a given index, that tuple with the bigger component on that index is the smaller tuple.
3. The components at a given index are compared. If the components at this index are equivalent, then the components with the next higher index are investigated. In the latter case the tuple with the smaller arity is also the smaller tuple.
4. 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 arity. Tuple sequences can be generated with tuple expressions:

[20] *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.

Additional Notation: The constant `null` represents the empty tuple sequence. It is typeless per se and is used for situations when no particular value should or can be used.

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

A *non-empty* tuple sequence is one which contains at least one tuple (that may or may not be empty).

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 defined in 4.8.2 is used.

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

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

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

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.

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 query expression will return a list of person names.

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

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

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

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

4.8.5 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.

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

4.9 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 ({ }).

```
[21] xml-content      ::= { xml-element }
[22] xml-element      ::= < xml-id { xml-attribute } xml-rest
[23] xml-id           ::= { (a-zA-Z) }
[24] xml-attribute    ::= xml-id = " { xml-fragment } "
[25] xml-rest         ::= > { xml-element | xml-fragment } </ xml-id      |
                          >
                          />
[26] xml-fragment     ::= xml-text | { query-expression }
[27] xml-text         ::= ...see text...
```

Text within attribute values may not include the string terminator, be it' or ". XML text within an element may not include terminators, < and { respectively. If the characters { and } are used as-is, they have to be encoded as &#007B; and &#007D;, respectively.

Embedded query expressions can only occur within an attribute value or as part of element text.

EXAMPLE:
The following XML content samples are valid:

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

The following XML content samples are invalid:

- `<copyright>Copyright Holder` (no end tag)
- `<mess{$x}>This is broken.</{$y}age>` (sub expressions not allowed inside tags)
- `<code lang="pascal">procedure TEST () { writeln; &#007D;.</code>` (opening `{` indicates subexpression)

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:

- A tuple sequence is iterated over according to its ordering, if such exists. A tuple is iterated over with increasing index.
- 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;`. 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[5]).

EXAMPLE:
Given the following code,

```
for $s in "Portishead"
for $x in <some>XML code</some>
return
  <message>{$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]:

[28] *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 """

  {$p = ! ! tm:id($p)}        # try to find subject locators first, take id otherwise
  {$p / name}                 # add all name characteristics
  homepage: http://company.org/{$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:

[29] *value-expression*	::=	*value-expression* *infix-operator* *value-expression*	|
		prefix-operator *value-expression*	|
		function-invocation	|
		content	
[30] *infix-operator*	::=	**...any in the predefined environment...**	
[31] *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 Clause 7. 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 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:

```
$person / age
$person / age >= 18
```

If a value expression computes directly content, that will result in a tuple expression (4.7). Otherwise, a

value expression is a function invocation (4.12).

4.12 Function Invocation

```
[32] function-invocation    ::= item-reference ( function-parameters )
[33] function-parameters    ::= tuple-expression | < identifier : value-expression >
```

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

The function arguments can be either assigned *positional* or *via names* (see 6.3.2):

1. For a *positional parameter association*, first the tuple expression is evaluated in the current context. It is an error if the result is a tuple sequence with a length other than `1`. The individual values of that tuple are then assigned to the formal parameters in the function `$0`, `$1`, etc. It is an error if there are less formal parameters than values in the tuple. The evaluation result of the function becomes the overall result.

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

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

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

   EXAMPLE:
   The function defined in 6.3.2 would be called as:

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

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

4.13 Boolean Expressions

4.13.1 Structure

WHERE clauses (6.5, 6.4) and filters (6.6) make use of boolean expressions:

```
[34] boolean-expression    ::= boolean-expression | boolean-expression |
                               boolean-expression & boolean-expression |
                               boolean-primitive
[35] boolean-primitive     ::= not boolean-primitive |
                               false |
                               ( boolean-expression ) |
                               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 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 ...
```

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** implicitely existentially quantified. This is meant as a safeguard to avoid that accidentially potentially large maps are traversed.

The constant false always evaluates to null, the empty sequence.

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

The operators &, | and not have the usual meaning (non-short-circuit, symmetric).

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:

[36] *exists-clause*	::=	**some**	*binding-set*	**satisfies**	*boolean-expression*

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

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

First, a sequence of bindings is generated. Then each of these bindings will be added to the current context for the evaluation of the boolean expression.

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

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

[O] *exists-clause*	::=	**exists**	*content*				
	==>	**some**	*%_*	**in**	*content*	**satisfies**	**not**	**false**

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: This can be further abridged to:

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

4.13.3 FORALL Clauses

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

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

Like for EXISTS clauses, a sequence of bindings is generated first. Each of these bindings are added to 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 added bindings 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 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 contain not a single politician, then this boolean expression would evaluate to true. Otherwise, only when each of them has (at least one) trait honorable, only then the expression is true.

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

[Q]	*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 (').

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

The sigil (either a $, @ or %) signals whether the variable can be bound to either an 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 types and their related functions from the *predefined environment* (Clause 7).

It is initially adopted from the querying application (explicitly or implicitely). 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() (7.2).

$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:

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

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

Variables always exist in a *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 modified 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 does actually bind.

EXAMPLE:
The following query expression finds all person's name(s) who live in a city; which city 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 implicitely let it range over all map items, such that the boolean expression can be satisfied. As such, every occurrence of $_ is *implicitely 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:

[39] *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 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 this 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:

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

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*.

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. Also null is used if a sequence contains more than one tuple. 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 will be pushed onto the context (5.4). Apart from this initial binding set, 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 it can be used by default.

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

During a query 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 environment.

Query evaluations may 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):

```
[41] 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 (Clause 7).

> 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 context map (the current value of%_) and the environment map (the current value of%__).

6.3 Environment Clause

6.3.1 Structure

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,
- functional dependencies (*functions*), and
- general ontological constraints and rules (*predicates*).
- external ontologies (*vocabularies*, *taxonomies*, *PSI sets*),

The *environment clause* allows to declare all of the above as part of a topic map (4.10):

```
[42] environment-clause    ::=  tm-content
```

EXAMPLE:
As any information in the environment map will be effectively merged with that in the queried map, it can be used add missing information to make queries more robust or concise.

In the following, a topic map about literature references is using concepts, such as *article*, *paper* or *proceedings*, but does not contain the information that all of these are actually subclasses of a *document*. To query for all documents, the query would have to list all these classes and would have to merge the partial results.

A much more concise solution is to make this subclassing explicit and use it in the query expression:

```
"""
  article    iko document

  paper      iko document

  proceedings iko document
"""
select $d / title
where
  $d isa document
```

A processor will recognize the following topic classes:

1. tmql:function: For every topic of such a class, a processor will register a *function* which can be invoked (4.11);
2. tmql:predicate: For every topic of such a class, a processor will register a *predicate* (6.3.3) which can be referred to;
3. tmql:ontology: For every topic of this class, the item identifier will be locally registered 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:
As 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) which operate on tuple sequences, one can use:

```
select $f / name, $f / description
from %__
where
   $f isa tmql:function
  & tmql:operates-on (tmql:function: $f, tmql:type: tmql:tuple-sequence)
```

6.3.2 Functions

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

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

```
nr-accounts is-a tmql:function
tmql:return: "
  fn:length ( %map // holds-account ( . -> holder == $user ) )
"
```

TMQL itself can be used to implement a function body. In this case the function topic must have exactly one ocurrence of type tmql:return and its value must be a string following the *content* syntax (4.7).

NOTE:
Functions cannot be overloaded.

Once a function topic is registered in the environment map, the function itself can be invoked (4.12) whereever this environment map is used. There, it is addressed via its item identifier.

If the function should be invoked via *positional parameter association*, then inside the function body the special variables $0, $1, etc. must be used.

If the function should be invoked via *named parameter association*, then the formal parameters are determined implicitely to be the names of all free variables within the function body, i.e. those variables which are not quantified in FOR, SOME or FORALL clauses.

EXAMPLE:
In the function body `fn:length ( %map // holds-account ( . -> holder == $user ) )` the free variables are `%map` and `$owner`, then the formal parameters are `map` and `owner`, respectively. If such function is to be invoked inside a query expression, then actual parameters for these two variables have to be provided:

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

6.3.3 Predicates

Boolean expressions (4.13) can be interpreted as anonymous predicates. Depending on the variable bindings in the context, that anonymous predicate either returns a non-empty or an empty sequence, answering the question whether particular values in a map are in a particular relationship or not. Predicates can be used to constrain relationships between values and can model certain aspects of application domain.

Boolean expressions can be named by declaring a topic of type `tmql:predicate` in the environment map.

EXAMPLE:
The following declares the predicate `is-edutainer-at`:

```
is-edutainer-at isa tmql:predicate
description: defines an entertaining academic somewhere
tmql:where: """
    $person isa person
  & $uni    isa university
  & $person / trait == "funny"
  & is-employed-at (employee: $person, employer : $uni)
"""
```

Predicates must have exactly one occurrence of type `tmql:where` with a string value following itself the syntax of boolean expressions (4.13).

To determine the formal parameters of the predicate, all free variables (again ignoring `$_`s) are determined in the boolean expression. Then the sigil is dropped from the variable names.

EXAMPLE:
The free variables in the above predicate were `$person` and `$uni`; therefore the formal parameters are `person` and `uni`.

NOTE:
Predicates cannot be overloaded.

A processor will interpret a named predicate as abstraction for the set of associations (*virtual associations*) which can be generated according to the following procedure:

1. All free variables in the predicate body will be existentially quantified, i.e. will range over all items in the context map.
2. For each such binding set generated, the predicate body will be evaluated.
3. If the result is an empty sequence this binding set will be discarded.
4. For each of the remaining binding sets an association with the following structure is generated:
    1. As association type the predicate item is used.
    2. For every binding in the binding set a role is built whereby the name of the variable is used as the item identifier for the role and the value of the binding is used for the player.
    3. As scope the universal scope is used.

5. The procedure is repeated until no new associations can be generated (fixpoint semantics).

All virtual associations are regarded to be merged into the effective map. Consequently, all navigational moves (4.4) have to be respect these associations, not only those which natively reside in the context map.

EXAMPLE:
The following expression finds all edutainers in Australian universities:

```
select $p
where
  is-edutainer-at (person: $p, uni: $u)
 & is-located-in   (object: $u, location: australia)
```

The obvious expectation here is that both predicates have to be satisfied, i.e. both have to correspond to associations, whether these are part of the instance data in the context map, or whether they are *derived* via a predicate.

Such derived knowledge not only applies to complete associations, but also to simple navigational moves. The following query expression looks for all things which are involved as person in an association of type is-edutainer-at.

```
select $p
where
  $p <- person ( ^ is-edutainer-at )
```

NOTE:
Implementations can choose to generate these associations at any time they find suitable.

NOTE:
Predicates can be recursive, directly or indirectly.

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

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

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

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

As the predicate is a (virtual) association, it can also be used for navigation:

```
%_ // city ( . <- location -> tm:subject == eiffel-tower )
```

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

6.3.4 Ontologies

Topics of type tmql:ontology in the environment map are interpreted as representatives of an ontology. Processors are free to support any available ontology definition technology. It is not specified by this International Standard how implementations determine which technology is to be used. The only constraint is that any ontology referenced can be experienced as topic map.

NOTE:
Ontologies might be simple vocabularies, they might be only a taxonomy (vocabulary embedded into a type system; or they might be full ontologies with (onto)logical rules, axioms or predicates. Obvious candidates for languages to define ontological constraints are TMCL (TMCL[6]) and OWL (OWL[7]).

These newly imported ontologies can be referenced directly, can be indicated by a subject identifier, or can be provided inline, as part of the topic.

> In any case, the item identifier of every ontology topic becomes a*prefix*, so that that can be used within a QName (4.2).

EXAMPLE:
The following query looks for all operas composed by Verdi:

```
"""
  wp isa tmql:ontology ~ http://en.wikipedia.org/wiki/
"""
select $opera
where
   is-composed-by (composer: wp:Verdi, opus: $opera)
 & $opera isa wp:Opera
```

The referenced ontology (Wikipedia) is only used to provide a convenient prefix for the namespace http://en.wikipedia.org/wiki/. The two QNames, wp:Verdi and wp:Opera, are expanded to full IRIs according to 4.3 which are used as subject identifiers. This assumes that the same subject identifiers are used in the context map.

Processors are free to recognize natively any reference to particular ontologies and attach any suitable semantics to them.

EXAMPLE:
The following declaration will introduce OpenCyc as ontology. A processor may recognize the URL and may load a library procuring the OpenCyc vocabulary.

```
"""
  cyc isa tmql:ontology ~ http://www.opencyc.org/
"""
```

Processors can also attach additional semantics to certain vocabularies.

EXAMPLE:
In the following environment map, two new prefixes are created:

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

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

The first one, ltm stands for the vocabulary as defined by LTM (Linear Topic Map Notation,LTM[3]). A processor may recognize this vocabulary natively, i.e. it may be able to parse LTM text streams.

The second topic is not only marked as ontology, but also asltm:instance, so that a processor is informed that it would have to deserialize the content fromfile:operas.ltm as LTM stream; if that is needed.

As opera is a topic representing the whole ontology it can be used as navigation starting point to follow the reification (4.4, reifier axis):

```
select ...
from %_ ++ opera ~~>
```

That will instruct a processor to actually read the file and parse LTM text. The result of this will be merged with the current context map. Not only is the context map enriched with all concepts in the

opera ontology, also inside the query expression prefixed references, such as `opera:libretto` can be used.

Ed. Note:
The following example is highly speculative.

EXAMPLE:
The following environment map references the vocabulary of the AsTMa= notation. It then uses that very notation to denote inline an adhoc vocabulary about operas:

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

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

    is-libretto isa tmql:predicate
    tmql:where : "
       # ... condition on $opus ....
    "
  """

"""
```

Like in the previous example, this ontology could be merged with the current context map. It also can remain separate if only functions or predicates are used from it:

```
select $text / name
where
  $text isa text
 & opera:is-libretto (opera:opus : $text)
```

6.4 SELECT Expressions

Query expressions can take the form of SELECT expressions:

[43] *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 false` 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.

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 (yet unbound) variables in the SELECT clause and do not constrain them in a WHERE clause are erroneous. This is meant as a safeguard to avoid that queries accidentially traverse — potentially huge — topic maps.

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

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

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

```
select $thing from %map
  where $thing isa 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.

If no UNIQUE clause exists, then the tuple sequence computed above remains unchanged in this step. Otherwise the function `fn:unique` (Clause 7) will be applied to it, rendering a new tuple sequence where all duplicate tuples have been eliminated.

Finally, the tuple sequence is subjected to a further function, `fn:slice` (Clause 7), 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 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:

```
[44] 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 false` is assumed.

A given FLWR expression can contain any number of *FOR clauses*. Several variable associations can be listed in a particular *FOR clause*. This is equivalent to having a dedicated FOR clause for every individual variable. Any introduced variable is visible until the end of this 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 persons are members within the same group (whereby the group itself is ignored):

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

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

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

After that one binding set has been added to the context, 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.

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

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

6.6 Path Expressions

6.6.1 Structure

Path expressions follow a *navigate and filter* 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.

[45] *path-expression*	::=	(*tuple-expression* | *simple-content*) { *postfix* }
[46] *postfix*	::=	*predicate-postfix* | *projection-postfix*

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

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

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

As starting point it uses the map bound to %biblio. The postfix (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 the .) 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:

[47] *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:

[S] *predicate-postfix*	::=	// *item-reference*
	==>	**[** ^ *item-reference* **]**

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

```
%map // person
```

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

[T] *path-expression*	::=	// *item-reference* { *postfix* }
	==>	% // *item-reference* { *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:

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

EXAMPLE:
To select the first 10 persons from a tuple sequence, one can use:

```
// person ( 0 .. 10 )
```

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

```
[W] boolean-primitive      ::=   ^ item-reference
                           ==>   . >> classes == item-reference
[X] boolean-primitive      ::=   @ item-reference
                           ==>   . @ == item-reference
```

EXAMPLE:
To select only the english names from persons, one can use:

```
// person / name ( @ english )
```

6.6.3 Projection Postfix

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

```
[48] 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 operas 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 operas in the map.

EXAMPLE:
The path expression

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

extracts first all instances of operas. Then a new tuple is (maybe, see below) 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 Association Predicate Invocations

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

```
[Y] path-expression      ::=  item-reference ( < item-reference :
                              value-expression > [ , ... ] )
                         ==> // item-reference
                              [ . / item-reference-1 == value-expression-1
                              ]
                              [ . / item-reference-2 == value-expression-2
                              ]
                               ...see text...
```

NOTE:
While predicate invocations are a more concise notation for their purpose than general path expressions, they do not introduce new expressitivity. Their (informal) semantics is described below as it cannot be fully described by a syntactical transformation.

The item-reference before the opening bracket is interpreted to be an association type in the context map. Only such associations in the context map are considered to be in the result, which are a (direct or indirect) instance of this type.

Any number of role/player combinations can be specified. The roles are all item identifiers, the players are computed via a value expression. The optional ellipsis ... can be used to indicate that matching associations may contain other role/player not constrained by the invocation.

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

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

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

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

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

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

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 topic items as values; otherwise an error is flagged. For one role this is considered as list of *potential players*.

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

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

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

```
[Z]  path-expression     ::=  simple-content-1 iko simple-content-2
                          ==> tm:subclass-of ( tm:subclass :
                              simple-content-1 , tm:superclass :
                              simple-content-2 )
[AA] path-expression     ::=  simple-content-1 is-a simple-content-2
                          ==> tm:type-instance ( tm:instance :
                              simple-content-1 , tm:type : simple-content-2
                              )
```

7 Predefined Environment

7.1 TMQL Concepts

This map defines the main concepts of TMQL.

```
# defining the target namespace

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

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

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

function is-a concept
! name: Function
description: computes a new value from existing ones
comment    : there are predefined functions and user-defined ones

predicate is-a concept
! name: Predicate
description: stands for a collection of associations

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

datatype is-a concept
! name: Data Type

primitive-datatype iko datatype
! name: Primitive Data Type

boolean is-a primitive-datatype ~ http://www.w3.org/TR/xmlschema-2/datatypes.html#boolean
! name: Boolean

integer is-a primitive-datatype ~ http://www.w3.org/TR/xmlschema-2/datatypes.html#integer
! name: Integer Number

decimal is-a primitive-datatype ~ http://www.w3.org/TR/xmlschema-2/datatypes.html#decimal
! name: Decimal Number

date    is-a primitive-datatype ~ http://www.w3.org/TR/xmlschema-2/datatypes.html#dateTime
! name: Datetime

uri     is-a primitive-datatype ~ http://www.w3.org/TR/xmlschema-2/datatypes.html#anyURI
! name: uri

string  is-a primitive-datatype ~ http://www.w3.org/TR/xmlschema-2/datatypes.html#string
! name: String

complex-datatype iko datatype
! name: Complex Data Type

xml   isa complex-datatype
! name: XML Content
```

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

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

7.2 Types and Functions

This map contains the detailed definitions of the used data types, their syntax and all related functions.

```
@@@@ here we have to decide which data type we actually want/need
```

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 and rejects all other streams with flagging errors at those situations described in this International Standard.
2. It provides a minimal functional environment as defined in Clause 7. 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 of this data type. They also have to specify stringification rules.
3. Processors may provide apriori knowledge of particular ontologies. The only way they are allowed to recognize this is via IRIs used as subject identifiers for these ontologies.
4. It delivers all evaluation results for valid query expressions according to the (formal) semantics Clause 9. Processor may deliver results eagerly or lazily.

NOTE:
The inferencing mechanics cannot be extended.

9 Formal Semantics

9.1 Mapping Association Predicates to Path Expressions

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

A Delimiting Symbols (normative)

Following characters are delimiting:

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

B Syntax (informative)

Core Syntax

[17]	*anchor*	::=	*constant* | *variable*
[2]	*atom*	::=	*boolean* |
			integer |
			decimal |
			iri |
			date |
			string [^^ *iri-or-qname*]

```
[15] axis              ::=  classes          |
                            superclasses     |
                            players          |
                            roles            |
                            characteristics  |
                            scope            |
                            locators         |
                            indicators       |
                            reifier          |
                            atomify
[40] binding-set       ::=  < variable-assignment >
[3]  boolean           ::=  true | false
[34] boolean-expression ::= boolean-expression | boolean-expression |
                            boolean-expression & boolean-expression |
                            boolean-primitive
[Q]  boolean-expression ::= every binding-set satisfies boolean-expression
                       ==> not some binding-set satisfies not (
                            boolean-expression )
[35] boolean-primitive ::=  not boolean-primitive |
                            false                 |
                            ( boolean-expression ) |
                            forall-clause         |
                            exists-clause
[W]  boolean-primitive ::=  ^ item-reference
                       ==> . >> classes == item-reference
[X]  boolean-primitive ::=  @ item-reference
                       ==> . @ == item-reference
[1]  constant          ::=  atom | item-reference
[19] content           ::=  content ( ++ | -- | == ) content |
                            { query-expression }             |
                            if path-expression then content [ else content ] |
                            tm-content                       |
                            xml-content
[L]  content           ::=  path_expression
                       ==> { path_expression }
[M]  content           ::=  path-expression-1 || path-expression-2
                       ==> if path-expression-1 then { path-expression-1 } else {
                            path-expression-2 }
[6]  date              ::=  ...xsd:dateTime...
[5]  decimal           ::=  /-?\d+(\.\d+)?/
[42] environment-clause ::= tm-content
[36] exists-clause     ::=  some binding-set satisfies boolean-expression
[O]  exists-clause     ::=  exists content
                       ==> some %_ in content satisfies not false
[P]  exists-clause     ::=  content
                       ==> exists content
[47] filter-postfix    ::=  [ boolean-primitive ]
[44] flwr-expression   ::=  { for binding-set }
                            [ where boolean-expression ]
                            [ order by < value-expression > ]
                            return content
[37] forall-clause     ::=  every binding-set satisfies boolean-expression
[32] function-invocation ::= item-reference ( function-parameters )
```

[33] *function-parameters* ::= *tuple-expression* | < *identifier* **:** *value-expression* >

[11] *identifier* ::= **/\w[\w\-\.]*/**

[30] *infix-operator* ::= **...any in the predefined environment...**

[4] *integer* ::= **/-?\d+/**

[7] *iri* ::= **"** *iri-or-qname* **"**

[8] *iri-or-qname* ::= **...xsd:anyURI...** | *qname*

[13] *item-reference* ::= *identifier* | *qname* | *iri*

[A] *item-reference* ::= *****

==> **tm:subject**

[J] *navigation* ::= **/** *item-reference* [*navigation*]

==> **>>** **characteristics** *item-reference* **>>** **atomify** [*navigation*]

[K] *navigation* ::= **** *item-reference* [*navigation*]

==> **<<** **atomify** **<<** **characteristics** *item-reference* [*navigation*]

[18] *navigation* ::= *step* [*navigation*]

[45] *path-expression* ::= (*tuple-expression* | *simple-content*) { *postfix* }

[T] *path-expression* ::= **//** *item-reference* { *postfix* }

==> **%_** **//** *item-reference* { *postfix* }

[Y] *path-expression* ::= *item-reference* **(** < *item-reference* **:** *value-expression* > [**,** **...**] **)**

==> **//** *item-reference*
[**.** **/** *item-reference-1* **==** *value-expression-1* **]**
[**.** **/** *item-reference-2* **==** *value-expression-2* **]**
...see text...

[Z] *path-expression* ::= *simple-content-1* **iko** *simple-content-2*

==> **tm:subclass-of** **(** **tm:subclass** **:** *simple-content-1* **,** **tm:superclass** **:** *simple-content-2* **)**

[AA] *path-expression* ::= *simple-content-1* **is-a** *simple-content-2*

==> **tm:type-instance** **(** **tm:instance** **:** *simple-content-1* **,** **tm:type** **:** *simple-content-2* **)**

[46] *postfix* ::= *predicate-postfix* | *projection-postfix*

[S] *predicate-postfix* ::= **//** *item-reference*

==> **[** **^** *item-reference* **]**

[U] *predicate-postfix* ::= **[** *integer* **]**

==> **[** **$#** **==** *integer* **]**

[V] *predicate-postfix* ::= **[** *integer-1* **..** *integer-2* **]**

==> **[** *integer-1* **<=** **$#** **&** **$#** **<** *integer-2* **]**

[10] *prefix* ::= **/\w+:/**

[31] *prefix-operator* ::= **...any in the predefined environment...**

[48] *projection-postfix* ::= *tuple-expression*

[9] *qname* ::= *prefix* *identifier*

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

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

[16] *simple-content* ::= *anchor* [*navigation*]

[14] *step* ::= (**>>** | **<<**) *axis* [*item-reference*]

```
[B]  step        ::=  >>  instances
                  ==> <<  classes
[C]  step        ::=  >>  subclasses
                  ==> <<  superclasses
[D]  step        ::=  ->  item-reference
                  ==> >>  players  item-reference
[E]  step        ::=  <-  item-reference
                  ==> <<  players  item-reference
[F]  step        ::=  @
                  ==> >>  scope
[G]  step        ::=  =
                  ==> <<  locators
[H]  step        ::=  ~
                  ==> <<  indicators
[I]  step        ::=  ~~>
                  ==> >>  reifier
[12] string      ::=  /"[^"]*"/  |  /'[^']*'/
[28] tm-content  ::=  """  ctm-instance  """
[20] tuple-expression  ::=  (  < value-expression [ asc | desc ] >  )
[N]  tuple-expression  ::=  null
                  ==> (  )
[29] value-expression  ::=  value-expression infix-operator value-expression  |
                            prefix-operator value-expression                  |
                            function-invocation                               |
                            content
[38] variable    ::=  /[$@%][\w#]+'*/
[R]  variable    ::=  .
                  ==> $0
[39] variable-assignment  ::=  variable  in  content
[24] xml-attribute  ::=  xml-id  =  "  { xml-fragment }  "
[21] xml-content  ::=  { xml-element }
[22] xml-element  ::=  <  xml-id  { xml-attribute }  xml-rest
[26] xml-fragment  ::=  xml-text | {  query-expression  }
[23] xml-id      ::=  { (a-zA-Z) }
[25] xml-rest    ::=  >  { xml-element | xml-fragment }  </  xml-id  >  |
                      />
[27] xml-text    ::=  ...see text...
```

Bibliography

TMQLreq, *TMQL Requirements*, ISO, 2003

TMQLuc, *TMQL Use Cases*, ISO, 2003

LTM, *LTM*, Ontopia, ,

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

XTM, *???*, , ,

TMCL, *???*, , ,

OWL, *???*, , ,