

# Topic Maps Query Language

2008-07-15

Lars Marius Garshol, Robert Barta

## Contents

- 1 [Scope](#)
- 2 [Normative references](#)
- 3 [Notation and Semantics](#)
- 3.1 [Syntax Conventions](#)
- 3.2 [Informal and Formal Semantics](#)
- 3.3 [Ontological Commitments](#)
- 4 [Content](#)
- 4.1 [Constants](#)
- 4.2 [Atoms](#)
- 4.3 [Item References](#)
- 4.4 [Navigation](#)
- 4.5 [Auto-Atomification](#)
- 4.6 [Simple Content](#)
- 4.7 [Composite Content](#)
- 4.8 [Tuples and Tuple Sequences](#)
- 4.8.1 [Tuples](#)
- 4.8.2 [Comparing Tuples](#)
- 4.8.3 [Tuple Expressions](#)
- 4.8.4 [Ordering Tuple Sequences](#)
- 4.8.5 [Stringifying Tuple Sequences](#)
- 4.9 [XML Content](#)
- 4.10 [Topic Map Content](#)
- 4.11 [Value Expressions](#)
- 4.12 [Function Invocation](#)
- 4.13 [Boolean Expressions](#)
- 4.13.1 [Structure](#)
- 4.13.2 [EXISTS Clauses](#)
- 4.13.3 [FORALL Clauses](#)
- 5 [Query Contexts](#)
- 5.1 [Variables](#)
- 5.2 [Variable Bindings](#)
- 5.3 [Implicit Existential Quantification](#)
- 5.4 [Variable Assignments](#)
- 5.5 [Binding Set Ordering](#)
- 6 [Query Expressions](#)
- 6.1 [Processing Model](#)
- 6.2 [Structure](#)
- 6.3 [Environment Clause](#)
- 6.3.1 [Directives](#)
- 6.3.2 [Pragmas](#)
- 6.4 [SELECT Expressions](#)
- 6.5 [FLWR Expressions](#)
- 6.6 [Path Expressions](#)
- 6.6.1 [Structure](#)

- 6.6.2 [Filter Postfix](#)
- 6.6.3 [Projection Postfix](#)
- 6.6.4 [Predicate Invocations](#)
- 7 [Formal Semantics](#)
- 7.1 [Notation](#)
- 7.2 [Semantics of Variables](#)
- 7.3 [Semantics of Content](#)
- 7.4 [Semantics of Query Contexts](#)
- 7.5 [Semantics of Query Expressions](#)
- 8 [Conformance](#)
- A [Predefined Environment](#)
- B [Delimiting Symbols](#)
- C [Syntax](#)

## 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 specifies syntactic rules to form valid query expressions to extract information from a Topic Maps instance and also provides an informal and a formal semantics for every syntactic form.

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

This International Standard does not define an API (application programming interface) to interact with query processors and also refrains from naming certain error conditions. 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, <http://www.isotopicmaps.org/sam/sam-model/>

**TMRM**, *ISO 13250-5 Topic Maps — Reference Model*, 2008, <http://www.isotopicmaps.org/tmrm/>

**CTM**, *ISO 13250-6 Topic Maps — Compact Syntax*, 2008-05-15, <http://www.isotopicmaps.org/ctm/ctm.html>

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

**XTM 2.0**, *Topic Maps — XML Syntax*, ISO 13250: Topic Maps, 2006-06-19, <http://www.isotopicmaps.org/sam/sam-xtm/>

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

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

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

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

## 3 Notation and Semantics

### 3.1 Syntax Conventions

The syntax is defined on three levels:

1. At the *token level* this International Standard makes use of regular expressions [[RegExp](#)] to specify case-sensitive character patterns for valid terminal symbols. They are enclosed in slashes (/ /) to differentiate them from constant tokens. The abbreviation `\w` is used 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 bold and are underlined.

Tokens not defined in the grammar are those for binary infix and unary prefix operators; these are specified in the predefined environment together with their function counterparts ([Annex A](#)).

As usual, character symbols are either *delimiting* or not. The list of delimiting symbols is provided in [Annex B](#). Any other symbol is not delimiting and whitespace characters (blank, tab and newlines) must be used for separation. Whitespace characters are allowed everywhere between two tokens; this is not encoded explicitly in the syntax. Whitespace characters are insignificant except within strings, XML fragments and within a CTM stream.

2. The *canonical syntax level* is defined using a context-free grammar ([[XML 1.0](#)]) with the following conventions: For `A *` we use `{ A }`, for `A ?` we use `[ A ]` and for `( A ( ' , ' A ) * ) ?` (a comma-separated, possibly empty list) we use `< A >`.

For alternatives we use `|` between the choices. The grammar is built in such a way so that the lexical order of choices resolves any ambiguities in that it always prefers constant tokens over regular expression tokens.

Productions for non-terminals are numbered for reference.

3. On top of the canonical syntax, a *non-canonical syntax level* is introduced to reduce the syntactic noise in actual query expressions. These abbreviations are defined via an additional grammar production whereby a term (a sequence of terminals and non-terminals) on the right-hand side of a production is expanded (using term substitution ==>) into another term. Any abbreviated form is semantically equivalent to its expanded form, so shortcuts do not add any computational complexity to the language but are only added for convenience. These mappings are numbered with letters for reference.

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

[Annex C](#) contains the complete language syntax. This grammar was produced for human consumption and is not optimized for a particular parsing method or for a minimum of non-terminals.

### 3.2 Informal and Formal Semantics

The semantics of TMQL is defined in prose throughout the text. Such prose 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 superseded by the formal semantics which is detailed in [Clause 7](#).

### 3.3 Ontological Commitments

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

1. For the binary relation between a subtype and a supertype the interpretation and representation in [\[TMDM\]](#) (7.3) is adopted and extended.  
Accordingly, for *transitivity* a TMQL processor will assume that if a concept B is a supertype of A, and C is a supertype of B, then also C is a supertype of A. A processor will also interpret such relation as *reflexive* so that every type is a subtype and a supertype of itself.  
For a *transitivity-free* operation any subclass relationship between concepts carries no inferred meaning.
2. For the binary relation between an instance and a type the interpretation and representation in [\[TMDM\]](#) (7.2) is adopted. Accordingly, a processor will assume that if a concept is an instance of a type C, that very concept is also an instance of all supertypes of C.  
For a *transitivity-free* operation any instance relationship between concepts only applies to these concepts.

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

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

#### tm

<http://psi.topicmaps.org/iso13250/model/>

This is the namespace for the concepts defined by TMDM (via the TMDM/TMRM mapping).

#### xsd

<http://www.w3.org/2001/XMLSchema#>

This is the namespace for the XML Schema Datatypes.

**tmql**

<http://psi.topicmaps.org/tmq1/1.0/>

Under this prefix the concepts of TMQL itself are located.

**fn**

<http://psi.topicmaps.org/tmq1/1.0/functions/>

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

**dc**

<http://purl.org/dc/terms/>

Under this prefix Dublin Core elements are located.

Each of these prefixes are manifested as topics in the *environment map* as described in [6.3.1](#).

## 4 Content

### 4.1 Constants

Constants are either atomic values from the sets of predefined basic data types ([Annex A](#)), or they can also be references to items in the current effective map ([6.2](#)):

[1] <i>constant</i>	::=	<a href="#">atom</a>   <a href="#">item-reference</a>
---------------------	-----	---

### 4.2 Atoms

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

[2] <i>atom</i>	::=	<a href="#">undefined</a>   <a href="#">boolean</a>   <a href="#">number</a>   <a href="#">date</a>   <a href="#">dateTime</a>   <a href="#">iri</a>   <a href="#">string</a> [ <a href="#">^^</a> <a href="#">QIRI</a> ]
[3] <i>undefined</i>	::=	<b>undef</b>
[4] <i>boolean</i>	::=	<b>true</b>   <b>false</b>
[5] <i>number</i>	::=	<a href="#">decimal</a>   <a href="#">integer</a>
[6] <i>decimal</i>	::=	<a href="#">/[+-]?[0-9]+(\.[0-9]+)?/</a>
[7] <i>integer</i>	::=	<a href="#">/[+-]?[0-9]+/</a>
[8] <i>date</i>	::=	... see <a href="http://www.w3.org/TR/xmlschema-2/#date">http://www.w3.org/TR/xmlschema-2/#date</a> , 3.2.9.1 Lexical representation ...
[9] <i>dateTime</i>	::=	... see <a href="http://www.w3.org/2001/XMLSchema#dateTime">http://www.w3.org/2001/XMLSchema#dateTime</a> , 3.2.7.1 Lexical representation...
[10] <i>iri</i>	::=	" <a href="#">QIRI</a> "
[11] <i>string</i>	::=	<a href="#">"/([^\"] \\")*/</a>   <a href="#">"/([^\'] \\')*/</a>

Obviously, decimal patterns are preferred over the shorter integer patterns. Strings may use either quotes (") or apostrophes (') as terminators. Any terminator can be escaped within the string with a leading backslash (\). Strings can reach over several lines. IRI literals are specialized strings; they must also be wrapped in quotes.

**EXAMPLE:**

The following are valid atoms (type in brackets):

```

23                # integer
3.1415           # decimal
"Hello World"    # string
'Dante\'s Hell'  # string
2005-10-16T10:29Z # date
"http://example.org/something" # iri
undef            # undefined

```

**EXAMPLE:**

The following are invalid atoms:

```

3,14              # comma instead of dot
- 273.15         # blank after the sign.

```

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

**EXAMPLE:**

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

```

"42"^^xsd:integer      # integer
"http://example.org/something"^^xsd:string # string
"http://example.org/something"^^xsd:anyURI # iri
"w3 Own u"^^what:ever  # what:ever

```

For references either absolute IRIs or QNames can be used:

[12] <i>QIRI</i>	::=	<a href="#">IRI</a>   <a href="#">QName</a>
[13] <i>IRI</i>	::=	<a href="#">/([^\&lt;'\{\} \^] - [#x00-#x20])*/</a>
[14] <i>QName</i>	::=	<a href="#">prefix identifier</a>
[15] <i>prefix</i>	::=	<a href="#">^w+:/</a>
[16] <i>identifier</i>	::=	<a href="#">^w[w\-\.\_]*/</a>

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

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

**NOTE:**

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

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

### 4.3 Item References

In the map to be queried (*effective map*, [6.2](#)) Topic Map items can be named directly via an *item identifier* or a *subject identifier*, as provided by an IRI or a *QName*. The latter is optionally wrapped into `<>`:

[17] <i>item-reference</i>	::=	<a href="#">identifier</a>   <a href="#">QIRI</a>   <code>≤</code> <a href="#">QIRI</a> <code>≥</code>
----------------------------	-----	--

1. | If the item reference is an *identifier* then this identifier is interpreted as an *topi item*

| *identifier* ([TMDM], Clause 5.1) for a topic in the effective map. The result is then this topic item; if no such topic exists, an error will be flagged.

**EXAMPLE:**

The following expression first identifies the topic with the item identifier `jack` and then retrieves all its names:

```
jack / name
```

**NOTE:**

While convenient, item references are not a robust way to identify topics. TMDM processors are not constrained how they assign item identifiers to items ([TMDM], Clause 5.1).

2. | If the item reference is a QName, then first that is expanded into an absolute IRI according to 4.2. That absolute IRI is interpreted as *subject indicator* for a topic in the effective map. If no such topic exist an error will be flagged.

**EXAMPLE:**

The item reference `http://example.org/something/` is interpreted as subject identifier.

**EXAMPLE:**

`tm:subject` can be used when the particular topic is not relevant, or unknown. As any Topic Map item represents an instance of *subject* by definition, this makes predicates such as `is-located-in (tm:subject : $place, location: paris)` more robust.

### 4.4 Navigation

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

[18] <i>step</i>	::= ( >>   << ) <i>axis</i> [ <i>anchor</i> ]
[19] <i>axis</i>	::= <b>types</b>
	<b>supertypes</b>
	<b>players</b>
	<b>roles</b>
	<b>traverse</b>
	<b>characteristics</b>
	<b>scope</b>
	<b>locators</b>
	<b>indicators</b>
	<b>item</b>
	<b>reifier</b>
	<b>atomify</b>

The optional anchor adds control information which is useful with some axes, but not others. If it is missing `tm:subject` will be assumed.

| When the anchor is evaluated, it must evaluate to a topic item and is interpreted as type. Then in all navigation steps the current setting for type transitivity (6.3.2) is honored.

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

**types**

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

**EXAMPLE:**

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

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

[A] <a href="#">step</a>	::= >>	<b>instances</b>
	==> <<	<b>types</b>

**NOTE:**

There is no navigation axis for deriving the data type of an atom, but the function `has_datatype` ([Annex A](#)) can be used.

**supertypes**

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

**EXAMPLE:**

`person >> supertypes` produces all supertypes of the concept `person`, say, `human`, `mammal`, etc. depending on the used taxonomy.

Additional Notation: Asking for all subtypes of an item is the inverse of asking for all supertypes.

[B] <a href="#">step</a>	::= >>	<b>subtypes</b>
	==> <<	<b>supertypes</b>

**players**

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

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

**EXAMPLE:**

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

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

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

[C] <a href="#">step</a>	::= =>	<a href="#">anchor</a>
	==> >>	<b>players</b> <a href="#">anchor</a>
[D] <a href="#">step</a>	::= <=<	<a href="#">anchor</a>
	==> <<	<b>players</b> <a href="#">anchor</a>

**EXAMPLE:**

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

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

**roles**

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

If the value is a topic item, in backward direction this step computes all association items where that topic is the role type. Multiple uses of one topic as role in one association causes multiple results. The optional item identifier has no relevance.

**EXAMPLE:**

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

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

**traverse**

If the value is a topic item, in forward direction this step computes first all associations where the topic plays a role. The type of the associations is constrained by the optional item. The overall result of this navigation is then a sequence of all players of these associations, whereby the incoming topic is deducted once from that sequence. In backward direction the result sequence will always be empty.

**EXAMPLE:**

The following navigation finds all co-authors of `jack`:

```
jack >> traverse co-authors
```

Additional Notation: For traversing over associations of a certain type the following shortcut is introduced:

[E] <code>step</code>	::=	<code>&lt;=&gt;</code>	<code>anchor</code>
	==>	<code>&gt;&gt;</code>	<code>traverse anchor</code>

**EXAMPLE:**

The above example can be rewritten as `jack <-> co-authors`.

If the value is an association item, in backwards direction this step computes all players being an instance of the type given by the optional item. The result of this step is a sequence of all association items minus the incoming one.

**EXAMPLE:**

The following navigation finds all associations which share a topic of type `person` with `$a`:

```
$a << traverse person
```

**characteristics**

If the value is a topic item, in forward direction this step computes all names and occurrences of that topic which are subtypes of the optionally specified item. The result is a sequence of name and occurrence items.

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

**EXAMPLE:**

The following navigation finds all occurrences of `jack`:

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

**EXAMPLE:**

The following navigation finds all nicknames of `jack`:

```
jack >> characteristics nickname
```

**EXAMPLE:**

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

```
jack >> characteristics tm:subject
```

**scope**

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

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

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

[F] <a href="#">step</a>	::=	@
	==>	>> <b>scope</b>

#### 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 an IRI, in backward direction this step retrieves all topic items which have this IRI as subject locator. The optional item has no relevance.

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

[G] <a href="#">step</a>	::=	=
	==>	<< <b>locators</b>

#### EXAMPLE:

If an XTM instance would be stored in a file `file:/maps/dictators.xtm`, then the expression `file:/maps/dictators.xtm =` would identify the document itself, but only if a topic with that subject locator exists in the map. If not, the result will be empty.

#### indicators

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

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

[H] <a href="#">step</a>	::=	~
	==>	<< <b>indicators</b>

#### 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 that map itself. If a topic with such an subject identifier exists in the effective map, that topic item is the result.

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 within the effective map with that subject identifier.

#### item

If the value is a topic item, then in forward direction this steps finds one item identifier and produces a string. The optional item has no relevance.

#### NOTE:

Processors should provide a consistent selection of item identifiers during one evaluation.

Additional Notation: To retrieve the topic item identifier as string the following shortcut exists:

[I] <a href="#">step</a>	::=	!
	==>	>> <b>item</b>

If the value is a string, then in backwards direction this steps finds one topic item which has this string as item identifier. The optional item has no relevance.

#### reifier

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

If the value is an association, a name or an occurrence 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:

[J] <a href="#">step</a>	::=	<a href="#">~&gt;</a>
	==>	>> <b>reifier</b>

#### 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 in the FROM clause.

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

[K] <a href="#">step</a>	::=	<a href="#">&lt;~</a>
	==>	<< <b>reifier</b>

#### atomify

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

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

#### EXAMPLE:

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

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

#### EXAMPLE:

The following navigation finds all topics which have a `homepage` occurrence with a certain URL (they cannot be names as these cannot contain URLs):

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

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

[L] <a href="#">navigation</a>	::=	! <a href="#">anchor</a> [ <a href="#">navigation</a> ]
--------------------------------	-----	---

```

=> >> characteristics anchor >> atomify
      [ navigation ]

```

**EXAMPLE:**

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

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

```

[M] navigation ::= \ anchor [ navigation ]
=> << atomify << characteristics
      anchor [ navigation ]

```

**EXAMPLE:**

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

```
23 \ age
```

**EXAMPLE:**

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

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

**NOTE:**

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

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

## 4.5 Auto-Atomification

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

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

**EXAMPLE:**

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

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

Atomification is postponed until one of the following situations occur:

1. | The name or occurrence is about to be passed to the environment as part of the result.

**EXAMPLE:**

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

```
select $p / homepage
where
```

```
$p isa person
```

2. | The name or occurrence is about to be compared in the process of ordering ([asc](#) or [desc](#), [4.8.2](#)).

**EXAMPLE:**

The following query expression will return person name(s) and age(s), in age descending order:

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

3. | The name or occurrence is about to be passed into a function in the process of a function invocation ([4.12](#)).

**EXAMPLE:**

The following query expression will return person name(s) and their age which is computed from the birthday using a custom function [age](#):

```
select $p / name, age ($p / birthday)
where
  $p isa person
```

**NOTE:**

This rule includes the situation when the name or occurrence is about to be compared to an atomic value.

**EXAMPLE:**

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

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

4. | The name or occurrence is used inside an XML fragment.

**EXAMPLE:**

In the following, the names of each book will be atomified so that they can be inserted as text into the XML stream:

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

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

**EXAMPLE:**

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

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

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

  ""
```

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

**EXAMPLE:**

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

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

**EXAMPLE:**

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

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

## 4.6 Simple Content

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

[20] <i>anchor</i>	::= <a href="#">constant</a>   <a href="#">variable</a>
--------------------	---

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

[21] <i>simple-content</i>	::= <a href="#">anchor</a> [ <a href="#">navigation</a> ]
[22] <i>navigation</i>	::= <a href="#">step</a> [ <a href="#">navigation</a> ]

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

**EXAMPLE:**

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

**EXAMPLE:**

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

## 4.7 Composite Content

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

Content can be constructed in different ways:

[23] <i>content</i>	::= <a href="#">content</a> ( <a href="#">++</a>   <a href="#">--</a>   <a href="#">==</a> ) <a href="#">content</a>
	{ <a href="#">query-expression</a> }
	<b>if</b> <a href="#">path-expression</a> <b>then</b> <a href="#">content</a> [ <b>else</b>
	<a href="#">content</a> ]
	<a href="#">tm-content</a>
	<a href="#">xml-content</a>

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

(4.10).

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

```
[N] content ::= path expression
      ==> { path expression }
```

1. | If content is combined with one of the binary operators, first both content operands are evaluated in the current context. This results in two tuple sequences.

1. | If the operator is `==` then the resulting tuple sequence consists of exactly those tuples which exist in both operand tuple sequences (*AND semantics*). Tuples are compared according to [4.8.1](#).

**EXAMPLE:**

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

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

**EXAMPLE:**

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

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

2. | If the operator is `++` then the resulting tuple sequence consists of all tuples from both operand tuple sequences (*OR semantics*).

**NOTE:**

The overall result sequence may contain duplicates.

3. | If the operator is `--` then the resulting tuple sequence consists of exactly those tuples which exist in the left operand tuple sequence, but not in the right (*except semantics*). Tuples are compared according to [4.8.1](#).

**EXAMPLE:**

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

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

2. Content can also be unconditionally generated with a query expression ([Clause 6](#)) when that is wrapped inside `{}` brackets.

3. | When a content is conditional (if-then-else), the result depends on the condition path expression ([6.6](#)). If that is evaluated 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 a 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: The following shorthand allows to test for the existence of values and to use a default value otherwise:

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

**EXAMPLE:**

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

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

## 4.8 Tuples and Tuple Sequences

### 4.8.1 Tuples

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

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

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

**NOTE:**

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

**EXAMPLE:**

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

### 4.8.2 Comparing Tuples

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

When tuples are to be compared with each other 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 special values `asc` (for *ascending*) or `desc` (for *descending*). If that ordering tuple is not explicit, a tuple containing only `asc` values is assumed.

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

1. | The empty tuple is smaller than any other tuple.
2. | The comparison of non-empty tuples is done component-wise starting with index `0`.
  - | The components at a given index are compared. If the components at this index are equivalent, then the components with the next higher index are investigated. In the latter case the tuple with the smaller length is also the smaller tuple.
  - | If the order tuple has `asc` as value on a given index, that tuple with the smaller component on that index is also the smaller tuple. If the ordering tuple has `desc` as value on a given index, that tuple with the bigger component on that index is the smaller tuple.

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

**EXAMPLE:**

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

**EXAMPLE:**

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

### 4.8.3 Tuple Expressions

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

```
[24] 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 the 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 particular ordering.

**EXAMPLE:**

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

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

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

```
[P] tuple-expression ::= null
      ==> { }
```

### 4.8.4 Ordering Tuple Sequences

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

1. | If the tuple sequence is generated from a tuple expression in which an *order direction* (asc or desc) for a component is used, then that sequence will be ordered. For this purpose, components which do not have an order direction will be assumed to have asc. Then the ordering as defined in [4.8.2](#) is used.

**EXAMPLE:**

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

**EXAMPLE:**

The following expression finds first all instances of person; then for each of these, the combinations of their names and ages are generated: // person ( . / name , . / age desc )

The second component of the projection carries an order direction, so that of the first component will default to asc. The resulting tuple sequence will then be ordered,

first according to the name value; when there is a draw, then according to the age information, but that in descending order. Note, that this is done independently for each person.

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

**EXAMPLE:**

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

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

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

**EXAMPLE:**

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

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

#### 4.8.5 Stringifying Tuple Sequences

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

When a tuple is *stringified* its components are first converted into their textual representations. All these representations are then concatenated in the order of their index, separated by commas. Consequently the empty sequence will be stringified into the empty string.

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

#### 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 output dynamically. These expressions must be properly nested using a balanced pair of curly brackets (`{ }`).

[25] <i>xml-content</i>	::= { <a href="#">xml-element</a> }
[26] <i>xml-element</i>	::= ≤ <a href="#">xml-tag</a> { <a href="#">xml-attribute</a> } <a href="#">xml-rest</a>
[27] <i>xml-tag</i>	::= [ <a href="#">prefix</a> ] <a href="#">xml-fragments</a>
[28] <i>xml-attribute</i>	::= [ <a href="#">prefix</a> ] <a href="#">xml-fragments</a> ≡ " <a href="#">xml-fragments</a> "
[29] <i>xml-rest</i>	::= / ≥ ≥ { <a href="#">xml-element</a>   <a href="#">xml-fragments</a> } ≤! <a href="#">xml-tag</a> ≥
[30] <i>xml-fragments</i>	::= { <a href="#">xml-text</a>   { <a href="#">query-expression</a> } }
[31] <i>xml-text</i>	::= ...see text...

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

**NOTE:**

XML prefixes are completely independent from TMQL prefixes.

**EXAMPLE:**

The following XML content samples are valid:

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

The following XML content samples are invalid:

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

The following XML content may produce errors during evaluation:

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

**NOTE:**

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

**NOTE:**

XML namespaces can be declared with `xmlns` attributes and will become effective in the result.

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 (indicated below by `␣`) and blanks (indicated by `␣`) and is followed by another 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 the current context in no particular order. The content generated by these expressions is then embedded into the XML content, replacing the text of the query expressions (including the `{ }` bracket pair) according to the following embedding rules:

- Atomic content is converted into its string representation.
- String content is used as-is except that the special characters `&`, `"`, `<`, `>`, `'` are automatically encoded into the predefined entities `&amp;`, `&quot;`, `&lt;`, `&gt;`, `&apos;`, respectively.
- XML content is used as-is.
- Tuple sequences are converted into their string representation ([4.8.5](#)).
- TM content is serialized using XTM ([XTM 2.0](#)) according to following rules:
 

Topic items are serialized using the `<topic>` element (XTM, 4.5) whereby all identification information (subject address and subject identifiers), all types, all names and all occurrences are included.

Name and occurrence items are serialized using the `<name>` (XTM, 4.10) and `<occurrence>` element (XTM, 4.15), respectively, whereby all type, scope and value information is included.

Association items are serialized using the `<association>` element (XTM, 4.18) whereby type, the scope and all roles and players are included.

For any topic `id` or `topicRef` an appropriate topic item identifier has to be used.

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

## 4.10 Topic Map Content

TM instance data can be generated using CTM [CTM] as syntax:

```
[32] tm-content ::= "" ctm-instance ""
```

The content is controlled using a CTM text stream, whereby TMQL query expressions can be embedded.

When such TM content is evaluated, a TMDM instance will be generated according to the CTM deserialization rules. The result is a tuple sequence of singletons which contain only either a topic item or an association item.

### EXAMPLE:

The following example copies over the `person` topics together with a newly generated association for each `person` instance:

```
for $p in // person return ""
  {$p}           # embed the whole topic

  is-employed (org: big-corp, employee: {$p})
  ""
```

Embedded query expressions must be wrapped inside a `{ }` bracket and are only allowed at following particular positions within the CTM text stream:

1. Wherever a topic (CTM, 3.6 Topics), or association (CTM, 3.8 Associations) declaration is expected:

The result of the query expression must be a tuple sequence consisting of singleton tuples, otherwise an error is flagged. These singletons will be injected into the TMDM instance as follows:

1. every topic item will be injected together with all identity information (subject identifiers and subject locators), all names and all occurrences (any reification thereof are ignored),
2. every association item will be injected together with its type, the scope and all role items,
3. every string which follows the CTM `iri-ref` syntax (CTM, 3.3.5 IRI References) will be interpreted as subject identifier; a topic item will be injected using this very subject identifier,
4. every string with CTM `identifier` syntax (CTM, 3.3.6 Topic Identity) will be interpreted as item identifier and a topic item with such item identifier will be injected,
5. every other content is ignored.

### EXAMPLE:

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

  {$p}           # embed the whole topic

  {$p <- member} # embed every association

  {$p ~ }       # create topics with these subject identifiers
```

```

        {'mr-x'}
    """
    """ # create topic with this item identifier

```

2. Wherever a topic assignment (CTM, 3.7 Topic Tail) is expected:

The result of the embedded query expression must be a tuple sequence consisting of singleton tuples, otherwise an error is flagged. These singletons will be injected into the TMDM instance as follows:

1. | every name or occurrence item will be attached to the current topic,
2. | every other content is ignored.

**EXAMPLE:**

```

for $p in // person
return ""

?
{$p / name}
{$p / homepage}

# use CTM wildcard
# add all names
# add all homepage occurrences
"""

```

3. Wherever a CTM literal (CTM, 3.4 Literals) within an assignment (name or occurrence) is expected:

The result of the embedded query expression must be a tuple sequence consisting of singleton tuples, otherwise an error is flagged. Every such tuple value is stringified according to [4.8.5](#).

1. | When inside a name assignment, for every tuple value a name item is generated for the current topic, whereby the tuple value is used as string name.
2. | When inside an occurrence assignment, for every tuple value an occurrence item is generated for the current topic, where the tuple value is reinterpreted as CTM literal.

**NOTE:**

This guarantees that any data type information for literals is properly carried over into the generated instance.

**EXAMPLE:**

```

for $p in // person
return ""

?
name: {$p / homepage }

# use CTM wildcard
# add (possibly a number of) names
"""

```

4. Inside a CTM string (CTM, 3.4 Literals):

The tuple sequence resulting from evaluating the query expression is stringified according to [4.8.5](#). That string is interpolated into the CTM string.

**EXAMPLE:**

```

for $p in // person
return ""

?
homepage: http://{ $p / domain }/{ $p! }.html

# use CTM wildcard
"""

```

5. Wherever a topic identifier (CTM, 3.3.6 Topic Identity) is expected:

The result of the query expression must be a single singleton tuple, otherwise an error is flagged. If the value is a string that can be interpreted as item identifier, then that is used. If the value is a topic item, then an item identifier is used. Otherwise an error is flagged.

**EXAMPLE:**

```

for $p in // person return ""
is-employed (org: big-corp, employee: {$p})

```

""

## 4.11 Value Expressions

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

[33] <i>value-expression</i>	::=	<a href="#">value-expression infix-operator value-expression</a>   <a href="#">prefix-operator value-expression</a>   <a href="#">function-invocation</a>   <a href="#">content</a>
[34] <i>infix-operator</i>	::=	...any in the predefined environment...
[35] <i>prefix-operator</i>	::=	...any in the predefined environment...

Value expressions are either directly producing content, using the invocation of a function, or a combination of other value expressions using binary infix or unary prefix operators. The accepted operators, their symbols and their precedence are defined in [Annex A](#). Operators with higher precedence bind stronger. Operators with the same precedence level are interpreted to bind left-to-right.

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

### EXAMPLE:

The following are valid value expressions:

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

## 4.12 Function Invocation

[36] <i>function-invocation</i>	::=	<a href="#">item-reference parameters</a>
[37] <i>parameters</i>	::=	<a href="#">tuple-expression</a>   ( < <a href="#">identifier</a> : <a href="#">value-expression</a> > )

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

Every function must have formal parameters to which the actual parameters will be assigned. The formal parameters can be *positional*; in this case their names are *\$0*, *\$1*, etc. Or they can be explicitly *named*.

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

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

### EXAMPLE:

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

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

Should a person have several `age` occurrences, then the result list will contain the square root for each of these.

2. For a *named parameter association*, first all value expressions are evaluated in the current context. Also here the result is effectively a tuple sequence except that the components of individual tuples are not addressable via an index, but a name.
- For each such (named) tuple the function is invoked, whereby actual and formal parameters are associated via their name. While it is allowed that a formal parameter in the function does not get associated a value, it is an error if an actual parameter does not have a corresponding formal parameter.

**EXAMPLE:**

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

```
nr-accounts (owner: "James Bond")
```

3. In addition to the formal parameters, the effective map is passed as a special parameter to the function. How this is accomplished is not specified by this International Standard.

**NOTE:**

How the formal parameters are made known to the TMQL processor is not specified by this International Standard.

## 4.13 Boolean Expressions

### 4.13.1 Structure

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

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

Boolean expressions can be combined with the binary boolean operators `&` (AND) and `|` (OR). Boolean primitives can be negated. The brackets `()` can be used to override the usual precedence, namely that `&` binds stronger than `|` and that `not` binds stronger than `&`. FORALL clauses test whether a certain condition — again a boolean expression — is satisfied for all members of a particular sequence of values. In contrast, an EXISTS clause tests whether a condition is satisfied by at least one of the sequence members.

**EXAMPLE:**

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

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

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

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

**NOTE:**

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

Before a boolean expression is evaluated all free occurrences of the variable `$_` are *existentially quantified* according to [5.3](#). Then the boolean expression is evaluated in the

current context.

**NOTE:**

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

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

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

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

[Q]	<code>boolean-primitive</code>	<code>::=</code>	<code>^</code>	<code>anchor</code>
		<code>==&gt;</code>	<code>:</code>	<code>&gt;&gt;</code> <code>types</code> <code>==</code> <code>anchor</code>
[R]	<code>boolean-primitive</code>	<code>::=</code>	<code>@</code>	<code>anchor</code>
		<code>==&gt;</code>	<code>:</code>	<code>@</code> <code>==</code> <code>anchor</code>

**EXAMPLE:**

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

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

#### 4.13.2 EXISTS Clauses

An EXISTS clause allows to test whether a particular condition can be satisfied by either a minimal or a maximal number of tuples out of a given sequence:

[40]	<code>exists-clause</code>	<code>::=</code>	<code>exists-quantifier</code> <code>binding-set</code> <code>satisfies</code> <code>boolean-expression</code>
[41]	<code>exists-quantifier</code>	<code>::=</code>	<code>some</code> <code> </code> <code>at least</code> <code>integer</code> <code> </code> <code>at most</code> <code>integer</code>

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

**EXAMPLE:**

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

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

First a sequence of binding sets is generated (5.4). Then each of these sets will be pushed onto the current context for the evaluation of the inner boolean expression.

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

**EXAMPLE:**

The following expressions all return the empty sequence:

```

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

some                satisfies exists 1  # not a single binding

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

```

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

```

[S] exists-clause ::= exists content
    ==> some $ in content satisfies not null

```

**EXAMPLE:**

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

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

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

```

[T] exists-clause ::= content
    ==> exists content

```

**EXAMPLE:**

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

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

instead of the more canonical

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

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

**EXAMPLE:**

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

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

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

#### 4.13.3 FORALL Clauses

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

[42] forall-clause	::= <b>every</b> <a href="#">binding-set</a> <b>satisfies</b> <a href="#">boolean-expression</a>
--------------------	--

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

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

**EXAMPLE:**

The following expression tests whether all politicians are honorable persons.

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

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

**EXAMPLE:**

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

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

**NOTE:**

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

[U] <a href="#">boolean-expression</a>	::= <b>every</b> <a href="#">binding-set</a> <b>satisfies</b> <a href="#">boolean-expression</a>
	==> <b>not</b> <b>some</b> <a href="#">binding-set</a> <b>satisfies</b> <b>not</b> ( <a href="#">boolean-expression</a> )

## 5 Query Contexts

### 5.1 Variables

During the evaluation variables are used to bind values. A variable identifier must be prefixed by a *sigil* (either a \$, @ or %) and can be postfixed by any number of primes ('):

[43] variable	::= <a href="#">/\$@%][\w#]+*'</a>
---------------	------------------------------------

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

**EXAMPLE:**

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

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

**%% (current environment map)**

The environment map ([6.3](#)) contains all necessary background knowledge for a TMQL

processor. This includes all data types and their related functions from the *predefined environment* ([Annex A](#)).

It is initially adopted from the querying application (explicitly or implicitly). Query expressions can locally enrich this environment.

#### **\$\_ (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*.

#### **\$\$ (current position)**

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

#### **\$0, \$1, \$2, ... (positional variables)**

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

#### **\$\_ (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.

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:

[V] <a href="#">variable</a>	::=	.
	==>	<b>\$0</b>

#### **EXAMPLE:**

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

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

## 5.2 Variable Bindings

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

#### **EXAMPLE:**

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

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

During the course of the nested evaluation of a query expression a processor will maintain stack of binding sets, the *variable context* (short: context). Whenever variables are introduced, a corresponding set of bindings will be generated. These binding sets will be pushed onto the context for the duration of the evaluation of the inner expressions. At the end of that evaluation the last binding set is popped from the context.

| A processor will always maintain the following constraints on contexts:

1. | The *value of a particular variable* in the context is determined by a binding for that variable in that binding set which has been pushed last onto the context.

2. | If the variable names differ only in the number of primes, then their values MUST differ.
3. | Any two different variables may be bound to different or the same values.

**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 (tm:subject : $a,   tm:subject : $a')
  & is-neighbor-of (tm:subject : $a', tm:subject : $a'')
  & is-neighbor-of (tm:subject : $a'', tm:subject : $a)
```

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

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

### 5.3 Implicit Existential Quantification

Like any other variable the anonymous variable `$_` can be used as placeholder to bind to any value. It is, however, *write-only* in that its value can never be retrieved; any appearance of `$_` (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 `$_` appearance binds to, but only the fact that it actually does bind.

**EXAMPLE:**

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

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

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

**EXAMPLE:**

The WHERE clause above equivalently can be written as:

```
where
  $p isa person
  & some $_273 in // tm:subject 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

A variable assignment creates a sequence of bindings.

[44] *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 bind simple values, then a sequence of variable bindings is generated whereby in each of these the variable is bound to exactly one value in all tuples within the tuple sequence. If the tuple sequence is ordered, then the bindings will be ordered accordingly; also every tuple will be iterated through following increasing indices.
2. | If the variable sigil is `@`, so that the variable can only hold tuples, then — one by one — the tuples within the tuple sequence are bound to the variable for individual bindings. If the tuple sequence is ordered, the bindings will follow this order.
3. | If the variable sigil is `*` so that the variable can hold a complete tuple sequence, then the complete tuple sequence will be bound; only one binding will be generated.

**EXAMPLE:**

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

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

**EXAMPLE:**

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

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

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

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

**EXAMPLE:**

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

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

## 5.5 Binding Set Ordering

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

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

| 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 `undef` is used. If the result contains more than one tuple, any of these can be used.

| A binding set *B1* is said to *occur before* another, *B2*, if *B1*'s order 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 sequence is ordered according to the

values of `$p / name`. In this order the binding sets are 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 processing environment will pass in an *initial binding set*. How this is achieved is not constrained by this International Standard. This binding set forms the initial context (5.4). Apart from this no other information is imported.

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

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

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

Query evaluations return results into the processing 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):

[46] <i>query-expression</i>	::=	[ <a href="#">environment-clause</a> ]
		( <a href="#">select-expression</a>
		<a href="#">flwr-expression</a>
		<a href="#">path-expression</a> )

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 in the predefined environment ([Annex A](#)).

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

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

- the context map (the current value of `%_`),
- the environment map (the current value of `%`) of the directly enclosing query expression,

or that which the caller provided.

## 6.3 Environment Clause

The environment map contains background knowledge a processor will use in addition to the context map. In many cases additional ontological knowledge can be used to enrich the query processors' understanding of the application domain.

```
[47] environment-clause ::= { directive | pragma }
```

Any additional information added to the environment map only effects the directly following query expression.

### 6.3.1 Directives

Directives add implicit knowledge to the evaluation process.

```
[48] directive ::= prefix-directive
[49] prefix-directive ::= %prefix identifier QIRI
```

For *prefix directives* each of the declared prefixes are manifested as topics in the *environment map* (6.3) by using the prefix identifier as topic item identifier and the namespace QIRI as subject identifier. All such topics are made instances of `tmql:ontology` (Annex A).

### 6.3.2 Pragmas

A processing pragma controls modality of the evaluation process:

```
[50] pragma ::= %pragma identifier QIRI
```

This International Standard defines the following pragma identifiers and possible QIRIs:

1. `taxonomy`

For a QIRI `tm:transitive` any subclass relationship in the effective map is interpreted *transitively*, i.e. according to [TMDM] (7.3). This is also the default.

To turn off transitivity, the QIRI must be set to `tm:intransitive`.

This International Standard does not constrain how pragmas are manifested in the environment map.

## 6.4 SELECT Expressions

Query expressions can take the form of SELECT expressions:

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

Only the SELECT clause is mandatory. If the FROM clause is missing, then `from %_` is assumed, i.e. the current context map will be queried. If the WHERE clause is missing, then `where not null` is assumed. If the OFFSET clause is missing `offset 0` is assumed.

**EXAMPLE:**

The query below lists all composers which have composed an opera. The variable `$opera` is only used internally in the query, and none of the values bound to it will show up in the

final query result:

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

**EXAMPLE:**

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

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

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

Then the value expressions in the optional OFFSET and the LIMIT clauses are evaluated. Their results must be non-negative integers, or `null`. Otherwise an error will be flagged. These values will be bound to the variables `$_lower` and `$_limit`, respectively. If there was no LIMIT clause, `$_limit` remains unbound.

Then all free unbound variables in the WHERE clause (not those in the SELECT clause, and not any appearances 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:**

Accordingly, query expressions are erroneous, which name yet unbound variables in the SELECT clause do not constrain these inside the WHERE clause. This is meant as a safeguard to avoid that queries accidentally traverse — potentially huge — topic maps.

This is also the motivation to rule out the use of anonymous variables in SELECT clauses.

**EXAMPLE:**

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

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

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

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

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

**EXAMPLE:**

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

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

The individual persons are sorted by their age, given that they have such property. Note

that the individual names are NOT sorted.

With each context from the sequence of contexts, the boolean expression in the WHERE clause is evaluated (4.13). If the evaluation result is not the empty sequence, then that particular context will be used to evaluate the tuple expression within the SELECT clause. Otherwise, this particular context will be ignored.

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

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

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

## 6.5 FLWR Expressions

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

```
[52] flwr-expression ::= [ for binding-set ]
                        [ where boolean-expression ]
                        [ order by < value-expression > ]
                        return content
```

Only the RETURN clause is obligatory; with it the result content is generated. All the other clauses are optional. If the WHERE clause is missing, the default `where not null` is assumed. If the FOR clause is missing `for $_ in undef` is assumed.

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

### EXAMPLE:

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

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

### EXAMPLE:

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

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

The variable associations inside the FOR clauses are evaluated in lexical order, starting with the first. Every evaluation of a single variable association results in a sequence of variable

bindings for the given variable (5.2). One by one, these bindings are added to the current context with which the rest of the variable associations are evaluated. At the end of this process stands a sequence of contexts.

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

Each context will be subjected to a test provided by the boolean expression in the WHERE clause. If the evaluation renders a non-empty sequence, then that context will be kept; otherwise it will be discarded.

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

## 6.6 Path Expressions

### 6.6.1 Structure

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

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.

[53] <i>path-expression</i>	::= <a href="#">postfixed-expression</a>   <a href="#">predicate-invocation</a>
[54] <i>postfixed-expression</i>	::= ( <a href="#">tuple-expression</a>   <a href="#">simple-content</a> ) { <a href="#">postfix</a> }
[55] <i>postfix</i>	::= <a href="#">filter-postfix</a>   <a href="#">projection-postfix</a>

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

#### EXAMPLE:

The following path expression computes a table with two columns:

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

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

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:

```
[56] 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. 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:

```
[W] filter-postfix ::= // anchor
    ==> [ ^ anchor ]
```

**EXAMPLE:**

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

```
%map // person
```

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

```
[X] path-expression ::= // anchor { postfix }
    ==> %_ // anchor { postfix }
```

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

```
[Y] filter-postfix ::= [ integer ]
    ==> [ $# == integer ]
[Z] filter-postfix ::= [ integer-1 .. integer-2 ]
    ==> [ integer-1 <= $# & $# <= integer-2 ]
```

**EXAMPLE:**

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

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

**EXAMPLE:**

To select the first 10 persons from a tuple sequence, one can use: `// person [ 0 .. 10 ]`. Note that the value at the index with the upper bound is NOT part of the result.

### 6.6.3 Projection Postfix

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

```
[57] 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 concatenation 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 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 plays 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 association, then not a single tuple will be generated for that opera. This would also happen if that opera had no opus number at all.

**6.6.4 Predicate Invocations**

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

<pre>[58] predicate-       invocation      ::= <a href="#">anchor</a> { &lt; <a href="#">anchor</a> : <a href="#">value-expression</a> &gt; [ ,                           ... ] }</pre>
---

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

**EXAMPLE:**

The following predicate invocation identifies all cities which contain theatres:

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

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

**EXAMPLE:**

Role types cannot be omitted. But the role type can be `tm:subject` if it does not matter:

```
composed-by (composer: vivaldi, tm:subject : opera)
```

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

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

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

1. | The association item must be an instance of the given type.

2. | For each of the roles mentioned in the invocation, that association item must have a role which is a subtype (direct or indirect) of the role specified; and for that very role it must have a player out of the sequence of potential players.
3. | If the ellipsis ... has not been used, there must not exist further roles in the association item which are not constrained by the predicate invocation.

**NOTE:**

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

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

[AA]	<u>path-expression</u>	::=	<u>simple-content-1</u>	<b>iko</b>	<u>simple-content-2</u>
		==>	<b>tm:subclass-of</b>	(	<b>tm:subclass</b> : <u>simple-content-1</u> ;
					<b>tm:superclass</b> : <u>simple-content-2</u> )
[AB]	<u>path-expression</u>	::=	<u>simple-content-1</u>	<b>isa</b>	<u>simple-content-2</u>
		==>	<b>tm:type-instance</b>	(	<b>tm:instance</b> : <u>simple-content-1</u> ;
					<b>tm:type</b> : <u>simple-content-2</u> )

## 7 Formal Semantics

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

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

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

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

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

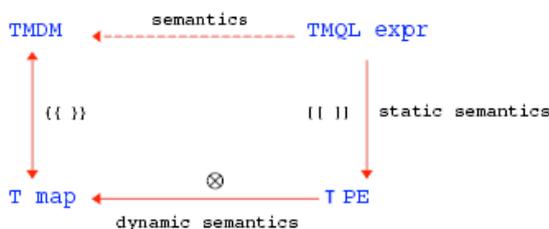


Figure 1 — Formal Semantics

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

denotes the inverse mapping from TMRM to TMDM.

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

$$\begin{aligned} & \text{eval}_{\text{TMDM}} ( \textit{query-expression} \mid B ) \\ & = \{ \{ \llbracket \textit{query-expression} \mid B \rrbracket \} \}^{-1} \end{aligned}$$

**NOTE:**

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

## 7.1 Notation

The formal semantics deals with several layers:

1. TMQL syntax layer: On this layer non-terminals are set in *italic* and terminals are **bold and underlined**.
2. Grammar layer: Here the symbols  $\{ \}, \langle \rangle, \llbracket \ \rrbracket, ()$  and  $|$  are used to express grammatical structures (3.1).
3. TMRM path expression level: Here the symbols are subscripted with  $\tau$ .

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

$$\begin{aligned} \llbracket \langle x \rangle \rrbracket &= \llbracket x_1, x_2, \dots, x_n \rrbracket \\ &= \llbracket x_1 \rrbracket, \llbracket x_2 \rrbracket, \dots, \llbracket x_n \rrbracket \\ &= \langle \llbracket x \rrbracket \rangle \end{aligned}$$

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

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

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

## 7.2 Semantics of Variables

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

$$\begin{aligned} & \llbracket p \mid B \rrbracket \\ &= \llbracket p\{[v^m \rightarrow u, v^n \rightarrow w]\} \perp \text{not } u \equiv w \perp \mid B\{[v^m \rightarrow u, v^n \rightarrow w]\} \rrbracket \end{aligned}$$

where  $u$  and  $v$  are newly introduced variables (not occurring anywhere in  $p$ ). Note that the substitution has to happen in  $p$  and also in all expressions in the variable assignments.

The process is repeated until no more primed variable pairs exist in  $p$ . The order of this elimination is not relevant.

- Any occurrences of  $\$$  within an expression  $p$  is replaced with a new variable not occurring in  $p$ :

$$\begin{aligned} & \llbracket p \mid B \rrbracket \\ &= \llbracket p\{[\underline{\$} \rightarrow u]\} \mid B \rrbracket \end{aligned}$$

Note that assignments are not affected.

- Named variables can be completely eliminated from expressions, so that only positional variables  $\$0, \$1, \dots$  remain in the down-translated expression. These positional variables directly correspond to projection:

$$\llbracket \$i \rrbracket = \pi_i$$

The elimination of named variables depends on their sigil:

- For variables with sigil  $\$$  (they can only hold simple values), an expression  $p$  under a binding  $\$v \Rightarrow q$  will be translated as follows:

$$\begin{aligned} & \llbracket p \mid \$v \Rightarrow q \rrbracket \\ &= \llbracket \text{fn:zigzag } (q) \rrbracket \\ & \otimes_{\top} \llbracket \perp \$0, p\{[\$v \rightarrow \$0]\} \perp \rrbracket \end{aligned}$$

First the expression of the binding is iterated through (using the predefined `fn:zigzag` function). Then the original expression is extended by a leading  $\$0$ . In the same step that expression is changed in that  $\$v$  is replaced with  $\$0$ , only after all instances of  $\$0$  are replaced with  $\$1$ , and that only after all instances of  $\$1$  are replaced with  $\$2$ , and so forth.

- For variables with sigil  $@$  (they can hold single tuples, i.e. lists), an expression  $p$  under a binding  $@v \Rightarrow q$  will be translated as follows:

$$\begin{aligned} & \llbracket p \mid @v \Rightarrow q \rrbracket \\ &= \llbracket q \rrbracket \\ & \otimes_{\top} \llbracket \perp \$0, \$1, \dots, \$n-1, p\{[@v \rightarrow (\$0, \$1, \dots, \$n-1)]\} \perp \rrbracket \end{aligned}$$

First the expression of the binding  $q$  is taken, then the original expression  $p$  is applied, but only after all occurrences of  $@v$  are replaced with the identity projection  $\$0, \$1, \dots, \$n-1$  with  $n$  the length of the tuples produced by  $q$ . All existing instances of  $\$i$  variables are again shifted to higher indices  $\$(i+n)$ .

- For variables with the sigil  $\%$  (they can hold tuple sequences), an expression  $p$  under a binding  $\%v \Rightarrow q$  will be translated as follows:

$$\begin{aligned} & \llbracket p \mid \%v \Rightarrow q \rrbracket \\ &= \llbracket p\{[\%v \rightarrow q]\} \rrbracket \end{aligned}$$

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

### 7.3 Semantics of Content

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

$$\llbracket \text{atom} \rrbracket = \{\{ \text{atom} \}\}$$

$$\llbracket \text{item-reference} \rrbracket = \{\{ \text{item-reference} \}\}$$

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

$$\llbracket \text{step} \rrbracket = \{\{ ( \geq | \leq ) \text{axis} \llbracket \text{anchor} \rrbracket \}\}$$

The only exception is the navigation step `traverse` which is not natively covered by the TMDM/TMRM mapping. For the forward direction we define

$$\llbracket \gg \text{traverse } \text{anchor} \rrbracket = \llbracket ( \text{\$0} \leftarrow \text{tm:subject} \llbracket \hat{\text{anchor}} \rrbracket \rightarrow \text{tm:subject} \text{ , } \text{\$0} \rrbracket \llbracket \text{not } \text{\$0} \equiv \text{\$1} \rrbracket \llbracket \text{\$0} \rrbracket \rrbracket$$

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

$$\begin{aligned} \llbracket \text{simple-content} \rrbracket &= \llbracket \text{anchor} \rrbracket \llbracket \otimes_{\top} \llbracket \text{navigation} \rrbracket \rrbracket \\ \llbracket \text{navigation} \rrbracket &= \llbracket \text{step} \rrbracket \llbracket \otimes_{\top} \llbracket \text{navigation} \rrbracket \rrbracket \end{aligned}$$

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

$$\llbracket \text{anchor} \rrbracket = \llbracket \text{constant} \rrbracket$$

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

$$\begin{aligned} \llbracket \text{content}_1 \text{ ++ } \text{content}_2 \rrbracket &= \llbracket \text{content}_1 \rrbracket \text{ + }_{\top} \llbracket \text{content}_2 \rrbracket \\ \llbracket \text{content}_1 \text{ -- } \text{content}_2 \rrbracket &= \llbracket \text{content}_1 \rrbracket \text{ - }_{\top} \llbracket \text{content}_2 \rrbracket \\ \llbracket \text{content}_1 \text{ == } \text{content}_2 \rrbracket &= \llbracket \text{content}_1 \rrbracket \text{ = }_{\top} \llbracket \text{content}_2 \rrbracket \end{aligned}$$

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

$$\llbracket \llbracket \text{query-expression} \rrbracket \rrbracket = \llbracket \text{query-expression} \rrbracket$$

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

$$\begin{aligned} &\llbracket \text{if } \text{path-expression} \text{ then } \text{content}_1 \text{ else } \text{content}_2 \rrbracket \\ &= \llbracket \text{path-expression} \rrbracket \text{ ? }_{\top} \llbracket \text{content}_1 \rrbracket \text{ : }_{\top} \llbracket \text{content}_2 \rrbracket \end{aligned}$$

Content can also be generated as TM fragment or as XML fragment. The static semantics is provided by  $\llbracket \text{tm-content} \rrbracket$  and  $\llbracket \text{xml-content} \rrbracket$ , respectively.

5. The static semantics of a tuple expression (4.8.3) is straightforward if no ordering is involved:

$$\begin{aligned} &\llbracket \llbracket \langle \text{value-expression} \rangle \rrbracket \rrbracket \\ &= ( \text{ }_{\top} \langle \llbracket \text{value-expression} \rrbracket \rangle \text{ } )_{\top} \end{aligned}$$

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

$$\begin{aligned} &\llbracket \llbracket \langle \text{value-expression}_i \text{ order}_i \rangle \rrbracket \rrbracket \\ &= ( \text{ }_{\top} \langle \llbracket \text{value-expression}_i \rrbracket \rangle \text{ } )_{\top} \otimes_{\top} \text{sort}_{\top}(\dots, \text{order}_i, \dots) \end{aligned}$$

Every `asc` in the ordering tuple is translated into  $\uparrow$  and every instance of `down` into  $\downarrow$ .

6. The static semantics for XML content (4.9) is provided by propagating  $\llbracket \rrbracket$  down the XML

fragment. The top-level elements are then treated as atomic values and are concatenated into a tuple sequence. Individual XML elements are collected and then are concatenated into an XML fragment.

$$\llbracket \{ \textit{xml-element} \} \rrbracket = \sum \{ \llbracket \textit{xml-element} \rrbracket \}$$

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

$$\begin{aligned} & \llbracket \leq \textit{xml-id} \{ \textit{xml-attribute} \} \textit{xml-rest} \rrbracket \\ = & \text{Element}_{\text{DOM}} (\textit{tagName} : \llbracket \textit{xml-id} \rrbracket, \\ & \textit{attributes} : \{ \llbracket \textit{xml-attribute} \rrbracket \} , \\ & \textit{childNodes} : \llbracket \textit{xml-rest} \rrbracket) \end{aligned}$$

and the element content is recursively built with:

$$\begin{aligned} & \llbracket \geq \{ \textit{xml-element} \mid \textit{xml-fragment} \} \rrbracket \\ = & \sum \{ \llbracket \textit{xml-element} \mid \textit{xml-fragment} \rrbracket \} \end{aligned}$$

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

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

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

Everywhere where a nested query expression is used within the CTM instance, that has to be mapped to  $\llbracket \textit{query-expression} \rrbracket$ . Since the CTM syntax is delegated, this is not further detailed here.

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

For a function invocation with positional parameter association, the static semantics is

$$\begin{aligned} & \llbracket \textit{item-reference} \textit{tuple-expression} \rrbracket \\ = & \{\{\textit{item-reference}\}\}_{\tau} (\llbracket \textit{tuple-expression} \rrbracket)_{\tau} \end{aligned}$$

For the named parameter association it is:

$$\begin{aligned} & \llbracket \textit{item-reference} \llbracket \textit{identifier}_i \textit{value-expression}_i \rrbracket \rrbracket \\ = & \{\{\textit{item-reference}\}\}_{\tau} (\llbracket \textit{value-expression}_i \rrbracket)_{\tau} \end{aligned}$$

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

$$\begin{aligned} & \llbracket \textit{boolean-expression}_1 \llbracket \textit{boolean-expression}_2 \rrbracket \rrbracket \\ = & (\llbracket \textit{boolean-expression}_1 \rrbracket)_{\tau} +_{\tau} (\llbracket \textit{boolean-expression}_2 \rrbracket)_{\tau} \end{aligned}$$

$$\begin{aligned} & \llbracket \textit{boolean-expression}_1 \ \& \ \textit{boolean-expression}_2 \rrbracket \\ = & (\llbracket \textit{boolean-expression}_1 \rrbracket)_{\tau} \otimes_{\tau} (\llbracket \textit{boolean-expression}_2 \rrbracket)_{\tau} \end{aligned}$$

Boolean primitives either nest properly:

$$\llbracket \llbracket \textit{boolean-expression} \rrbracket \rrbracket = (\llbracket \textit{boolean-expression} \rrbracket)_{\tau}$$

or they negate:

$$\begin{aligned} & \llbracket \textit{not} \textit{boolean-primitive} \rrbracket \\ = & \llbracket \textit{boolean-primitive} \rrbracket ?_{\tau} \llbracket 0_{\tau} \rrbracket :_{\tau} \llbracket (1) \rrbracket \end{aligned}$$

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

$$\begin{aligned} & \llbracket \text{some } binding\text{-set } \textit{satisfies} \textit{ } boolean\text{-expression} \rrbracket \\ & = \llbracket boolean\text{-expression} \rrbracket \mid \llbracket binding\text{-set} \rrbracket \end{aligned}$$

If the EXISTS clause has a lower bound  $N$ , then the semantics amounts to

$$\begin{aligned} & \llbracket \text{at least } integer \textit{ } binding\text{-set } \textit{satisfies} \textit{ } boolean\text{-expression} \rrbracket \\ & = \llbracket boolean\text{-expression}^{integer} \rrbracket \mid \llbracket binding\text{-set}^{integer} \rrbracket \end{aligned}$$

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

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

## 7.4 Semantics of Query Contexts

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

$$\llbracket \textit{variable } \textit{in} \textit{ } content \rrbracket = \textit{variable} \Rightarrow \llbracket content \rrbracket$$

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

$$\llbracket \langle \textit{variable-assignment} \rangle \rrbracket = \langle \llbracket \textit{variable-assignment} \rrbracket \rangle$$

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

$$\begin{aligned} & \langle \textit{variable-assignment} \rangle \\ & \textit{sorted-by} \langle \textit{value-expression} \llbracket \textit{asc} \mid \textit{desc} \rrbracket \rangle \end{aligned}$$

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

$$O = \langle \textit{value-expression} \rangle \llbracket \textit{O} \rrbracket$$

selecting only the first tuple in each of the intermediate results. The result tuple sequence  $s$  of the expression  $O \llbracket \langle \textit{variable-assignment} \rangle \rrbracket$  is then ordered according to the order direction tuple  $o = \llbracket \dots, \textit{asc}, \dots, \textit{desc}, \dots \rrbracket$  whereby

$$\begin{aligned} \llbracket \textit{asc} \rrbracket &= \uparrow \\ \llbracket \textit{desc} \rrbracket &= \downarrow \end{aligned}$$

The individual binding sets are then ordered according to the order produced by  $s \textit{sort}_T^O$  so that a binding set  $B$  is before another  $B'$  if the tuple produced with  $O \mid B$  comes before that of  $O \mid B'$ .

## 7.5 Semantics of Query Expressions

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

$$\begin{aligned} & \llbracket \textit{select} \langle \textit{value-expression}_1 \rangle \\ & \llbracket \textit{from} \textit{value-expression}_2 \rrbracket \\ & \llbracket \textit{where} \textit{boolean-expression} \rrbracket \\ & \llbracket \textit{order by} \langle \textit{value-expression}_3 \rangle \rrbracket \\ & \llbracket \textit{unique} \rrbracket \\ & \llbracket \textit{offset} \textit{value-expression}_4 \rrbracket \\ & \llbracket \textit{limit} \textit{value-expression}_5 \rrbracket \rrbracket \\ & = \llbracket \textit{for } \textit{\_} \textit{in} \textit{value-expression}_2 \\ & \textit{return} \{ \\ & \quad \textit{select} \langle \textit{value-expression}_1 \rangle \end{aligned}$$

```

    from %_
  [ where boolean-expression ]
  [ order by < value-expression3 > ]
  [ unique ]
  [ offset value-expression4 ]
  [ limit value-expression5 ]
} ]

```

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

```

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

```

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

```

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

```

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

```

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

```

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

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

```

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

```

```

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

```

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

```

[[ ( tuple-expression | simple-content ) { postfix } ]]
= [[ ( tuple-expression | simple-content ) ]] ⊗T { [[ postfix ] ] }

```

Postfixes themselves are either filters or projections:

```

[[ [ boolean-primitive ] ]]
= [[ boolean-primitive ]] ?T 1T :T [[ 0T ]]

```

```

[[ projection-postfix ]]
= [[ tuple-expression ]]

```

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

```

[[ anchor [ < anchori : value-expressioni > ] ] ]
= [[ // anchor < [ . -> anchori == value-expressioni ] > ]]

```

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

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

```

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

```

## 8 Conformance

A processor is conformant with this specification if:

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

**NOTE:**

Processors may provide a richer environment. Specifically, they may procure additional data types in which case they have to define equality, total ordering between values and optionally also operators for this data type. They also have to specify stringification rules to convert strings in the lexical value space to internal values and back.

3. It delivers all evaluation results for valid query expressions according to the (formal) semantics and it reports error conditions for all conditions defined here.

**NOTE:**

Processor may deliver results eagerly or lazily.

## A Predefined Environment (normative)

```

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

```

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

tmql-concept
- "TMQL core concept"

ontology isa tmql-concept
- "Ontology"

function isa tmql-concept
- "Function"

predicate iko function
- "Predicate"

binary-operator iko function
- "Binary Operator"

# every binary operator has a precedence value
# the higher the precedence, the closer it binds the operands
# left and right

unary-operator iko function
- "Unary Operator"

# unary prefix operators bind strongest

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

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

#-- data types -----

datatype isa tmql-concept
- "Data Type"

primitive-datatype iko datatype
- "Primitive Data Type"

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

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

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

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

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

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

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

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

xml isa primitive-datatype
- "XML Content"

complex-datatype iko datatype
- "Complex Data Type"

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

tuple-sequence isa complex-datatype
```

```

- "Tuple Sequence"
description: "a sequence of tuples -- can be ordered, or not"

#-- intrinsic functions

position isa function
  http://psi.topicmaps.org/tmql/1.0/functions/position
- "position"
profile: "fn:position()"
description: ""
Returns the index (natural number) of the tuple in the incoming
tuple sequence. If there is no such sequence, an error will be
flagged.
""

#-- functions and operators -----

#-- for decimals

decimal-unary-minus isa prefix-operator
  http://psi.topicmaps.org/tmql/1.0/functions/decimal-unary-minus
- "-" @ prefix-notation
profile: "fn:minus-decimal (a : decimal) return decimal"
precedence: 6

decimal-binary-add isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/decimal-binary-add
- "+" @ infix-notation
profile: "fn:add-decimal (a : decimal, b : decimal) return decimal"
precedence: 2

decimal-binary-minus isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/decimal-binary-minus
- "-" @ infix-notation
profile: "fn:minus-decimal (a : decimal, b : decimal) return decimal"
precedence: 2

decimal-binary-mul isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/decimal-binary-mul
- "*" @ infix-notation
profile: "fn:multiplication-decimal (a : decimal, b : decimal) return decimal"
precedence: 4

decimal-binary-div isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/decimal-binary-div
- "/" @ infix-notation
profile: "fn:division-decimal (a : decimal, b : decimal) return decimal"
precedence: 4

decimal-binary-mod isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/decimal-binary-mod
- "mod" @ infix-notation
profile: "fn:mod-decimal (a : decimal, b : decimal) return decimal"
precedence: 4

decimal-binary-lt isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/decimal-binary-lt
- "<" @ infix-notation
profile: "fn:lt-decimal (a : decimal, b : decimal) return tuple-sequence"
precedence: 5

decimal-binary-le isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/decimal-binary-le
- "<=" @ infix-notation
profile: "fn:leq-decimal (a : decimal, b : decimal) return tuple-sequence"
precedence: 5

decimal-binary-gt isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/decimal-binary-gt
- ">" @ infix-notation
profile: "fn:gt-decimal (a : decimal, b : decimal) return tuple-sequence"
precedence: 5

decimal-binary-ge isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/decimal-binary-ge
- ">=" @ infix-notation
profile: "fn:geq-decimal (a : decimal, b : decimal) return tuple-sequence"
precedence: 5

```

```

#-- for strings

string-concat isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/string-concat
- "+" @ infix-notation
profile: "fn:concat (a : string, b : string) return string"
precedence: 2

string-length isa function
  http://psi.topicmaps.org/tmql/1.0/functions/string-length
profile: "fn:length (s : string) return integer"

string-less-than isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/string-less-than
- "<" @ infix-notation
profile: "fn:string-lt (a : string, b : string) return tuple-sequence"
precedence: 5

string-less-equal-than isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/string-less-equal-than
- "<=" @ infix-notation
profile: "fn:string-leq (a : string, b : string) return tuple-sequence"
precedence: 5

string-greater-equal-than isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/string-greater-equal-than
- ">=" @ infix-notation
profile: "fn:string-geq (a : string, b : string) return tuple-sequence"
precedence: 5

string-greater-than isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/string-greater-than
- ">" @ infix-notation
profile: "fn:string-gt (a : string, b : string) return tuple-sequence"
precedence: 5

string-regexp-match isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/string-regexp-match
- "=~" @ infix-notation
profile: "fn:regexp (s : string, re : string) return tuple-sequence"
precedence: 5

#-- for tuple sequences

has-datatype isa function
  http://psi.topicmaps.org/tmql/1.0/functions/has-datatype
profile: "fn:has-datatype (s: tuple-sequence) return tuple-sequence"
description: ""
Retrieves for each tuple element in each tuple the data type. For
name items this is string, for occurrence item this is the data type
(not the type) of the occurrence value. For atoms this is the data
type of the atom. Each data type is a IRI.
""

has-variant isa function
  http://psi.topicmaps.org/tmql/1.0/functions/has-variant
profile: "fn:has-variant (s: tuple-sequence, s: item-reference) return tuple-sequence"
description: ""
Retrieves for each tuple element in each tuple the variant name for the given
scope. For name items this is the variant value, if such exists. Otherwise it
is undef. For all other things the function will return undef.
""

#--

slice isa function
  http://psi.topicmaps.org/tmql/1.0/functions/slice
profile: "fn:slice (s : tuple-sequence, low : integer, high : integer) return tuple-sequence"
description: "selects those tuples with index between low and high-1"

count isa function
  http://psi.topicmaps.org/tmql/1.0/functions/count
profile: "fn:count (s : tuple-sequence) return integer"
description: "returns the number of tuples in the sequence"

uniq isa function
  http://psi.topicmaps.org/tmql/1.0/functions/uniq

```

```

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

concat isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/concat
  - "++" @ infix-notation
profile: "fn:concat (s : tuple-sequence, t : tuple-sequence) return tuple-sequence"
description: "produces a tuple sequence with all tuples combined -- any ordering is honored"
precedence: 1

except isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/except
  - "--" @ infix-notation
profile: "fn:except (s : tuple-sequence, t : tuple-sequence) return tuple-sequence"
description: "produces a tuple sequence where all tuple which appear in t are removed from s"
precedence: 1

compare isa binary-operator
  http://psi.topicmaps.org/tmql/1.0/functions/compare
  - "==" @ infix-notation
profile: "fn:compare (s : tuple-sequence, t : tuple-sequence) return tuple-sequence"
description: "produces a tuple sequence of all tuples which appear in s and t"
precedence: 1

zigzag isa function
  http://psi.topicmaps.org/tmql/1.0/functions/zigzag
profile: "fn:zigzag (s : tuple-sequence) return tuple-sequence"
description: ""
returns a single tuple filled with all values from all tuples
index of tuples run faster than index within the tuple sequence
""

zagzig isa function
  http://psi.topicmaps.org/tmql/1.0/functions/zagzig
profile: "fn:zagzig (s : tuple-sequence) return tuple-sequence"
description: ""
returns a single tuple filled with all values from all tuples
index within the tuple sequence run faster than index within one tuple
""

```

## B Delimiting Symbols (normative)

Following characters are delimiting:

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

## C Syntax (informative)

### Core Syntax

[13]	<a href="#"><i>IRI</i></a>	::=	<u><a href="#">/([^\&lt;&gt;'{} ^] - [#x00-#x20])*/</a></u>	
[12]	<a href="#"><i>QIRI</i></a>	::=	<a href="#">IRI</a>   <a href="#">QName</a>	
[14]	<a href="#"><i>QName</i></a>	::=	<a href="#">prefix identifier</a>	
[20]	<a href="#"><i>anchor</i></a>	::=	<a href="#">constant</a>   <a href="#">variable</a>	
[2]	<a href="#"><i>atom</i></a>	::=	<a href="#">undefined</a>	
			<a href="#">boolean</a>	
			<a href="#">number</a>	
			<a href="#">date</a>	
			<a href="#">dateTime</a>	
			<a href="#">iri</a>	
			<a href="#">string</a> [ <sup>^^</sup> <a href="#">QIRI</a> ]	
[19]	<a href="#"><i>axis</i></a>	::=	<b>types</b>	
			<b>supertypes</b>	
			<b>players</b>	

		<b>roles</b>	
		<b>traverse</b>	
		<b>characteristics</b>	
		<b>scope</b>	
		<b>locators</b>	
		<b>indicators</b>	
		<b>item</b>	
		<b>reifier</b>	
		<b>atomify</b>	
[45]	<a href="#">binding-set</a>	::=	< <a href="#">variable-assignment</a> >
[4]	<a href="#">boolean</a>	::=	<b>true</b>   <b>false</b>
[38]	<a href="#">boolean-expression</a>	::=	<a href="#">boolean-expression</a>   <a href="#">boolean-expression</a>   <a href="#">boolean-expression</a> & <a href="#">boolean-expression</a>   <a href="#">boolean-primitive</a>
[U]	<a href="#">boolean-expression</a>	::=	<b>every</b> <a href="#">binding-set</a> <b>satisfies</b> <a href="#">boolean-expression</a> => <b>not</b> <b>some</b> <a href="#">binding-set</a> <b>satisfies</b> <b>not</b> ( <a href="#">boolean-expression</a> )
[39]	<a href="#">boolean-primitive</a>	::=	( <a href="#">boolean-expression</a> )   <b>not</b> <a href="#">boolean-primitive</a>   <a href="#">forall-clause</a>   <a href="#">exists-clause</a>
[Q]	<a href="#">boolean-primitive</a>	::=	^ <a href="#">anchor</a> => . >> <b>types</b> == <a href="#">anchor</a>
[R]	<a href="#">boolean-primitive</a>	::=	@ <a href="#">anchor</a> => . @ == <a href="#">anchor</a>
[1]	<a href="#">constant</a>	::=	<a href="#">atom</a>   <a href="#">item-reference</a>
[23]	<a href="#">content</a>	::=	<a href="#">content</a> ( ++   -   == ) <a href="#">content</a>   { <a href="#">query-expression</a> }   <b>if</b> <a href="#">path-expression</a> <b>then</b> <a href="#">content</a> [ <b>else</b> <a href="#">content</a> ]   <a href="#">tm-content</a>   <a href="#">xml-content</a>
[N]	<a href="#">content</a>	::=	<a href="#">path expression</a> => { <a href="#">path expression</a> }
[O]	<a href="#">content</a>	::=	<a href="#">path-expression-1</a>    <a href="#">path-expression-2</a> => <b>if</b> <a href="#">path-expression-1</a> <b>then</b> { <a href="#">path-expression-1</a> } <b>else</b> { <a href="#">path-expression-2</a> }
[8]	<a href="#">date</a>	::=	... see <a href="http://www.w3.org/TR/xmlschema-2/#date">http://www.w3.org/TR/xmlschema-2/#date</a> , 3.2.9.1 Lexical representation ...
[9]	<a href="#">dateTime</a>	::=	... see <a href="http://www.w3.org/2001/XMLSchema#dateTime">http://www.w3.org/2001/XMLSchema#dateTime</a> , 3.2.7.1 Lexical representation...
[6]	<a href="#">decimal</a>	::=	/[+]?d+(\.d+)?/
[48]	<a href="#">directive</a>	::=	<a href="#">prefix-directive</a>
[47]	<a href="#">environment-clause</a>	::=	{ <a href="#">directive</a>   <a href="#">pragma</a> }
[40]	<a href="#">exists-clause</a>	::=	<a href="#">exists-quantifier</a> <a href="#">binding-set</a> <b>satisfies</b> <a href="#">boolean-expression</a>
[S]	<a href="#">exists-clause</a>	::=	<b>exists</b> <a href="#">content</a> => <b>some</b> \$ <b>in</b> <a href="#">content</a> <b>satisfies</b> <b>not</b> <b>null</b>
[T]	<a href="#">exists-clause</a>	::=	<a href="#">content</a> => <b>exists</b> <a href="#">content</a>
[41]	<a href="#">exists-quantifier</a>	::=	<b>some</b>   <b>at least</b> <a href="#">integer</a>   <b>at most</b> <a href="#">integer</a>
[56]	<a href="#">filter-postfix</a>	::=	[ <a href="#">boolean-primitive</a> ]
[W]	<a href="#">filter-postfix</a>	::=	// <a href="#">anchor</a> => [ ^ <a href="#">anchor</a> ]
[Y]	<a href="#">filter-postfix</a>	::=	[ <a href="#">integer</a> ]

	<code>==&gt; [ \$# == integer ]</code>
[Z] <a href="#">filter-postfix</a>	<code>::= [ integer-1 .. integer-2 ]</code>
	<code>==&gt; [ integer-1 &lt;= \$# &amp; \$# &lt; integer-2 ]</code>
[52] <a href="#">flwr-expression</a>	<code>::= [ for binding-set ]</code>
	<code>[ where boolean-expression ]</code>
	<code>[ order by &lt; value-expression &gt; ]</code>
	<code>return content</code>
[42] <a href="#">forall-clause</a>	<code>::= every binding-set satisfies boolean-expression</code>
[36] <a href="#">function-invocation</a>	<code>::= item-reference parameters</code>
[16] <a href="#">identifier</a>	<code>::= <math>\wline\wline\wline!</math></code>
[34] <a href="#">infix-operator</a>	<code>::= ...any in the predefined environment...</code>
[7] <a href="#">integer</a>	<code>::= <math>[+]?d+</math></code>
[10] <a href="#">iri</a>	<code>::= " QIRI "</code>
[17] <a href="#">item-reference</a>	<code>::= identifier   QIRI   ≤ QIRI ≥</code>
[L] <a href="#">navigation</a>	<code>::= / anchor [ navigation ]</code>
	<code>==&gt; &gt;&gt; characteristics anchor &gt;&gt; atomify [ navigation ]</code>
[M] <a href="#">navigation</a>	<code>::= \ anchor [ navigation ]</code>
	<code>==&gt; &lt;&lt; atomify &lt;&lt; characteristics anchor [ navigation ]</code>
[22] <a href="#">navigation</a>	<code>::= step [ navigation ]</code>
[5] <a href="#">number</a>	<code>::= decimal   integer</code>
[37] <a href="#">parameters</a>	<code>::= tuple-expression   ( &lt; identifier : value-expression &gt; )</code>
[53] <a href="#">path-expression</a>	<code>::= postfixed-expression   predicate-invocation</code>
[X] <a href="#">path-expression</a>	<code>::= // anchor { postfix }</code>
	<code>==&gt; %_ // anchor { postfix }</code>
[AA] <a href="#">path-expression</a>	<code>::= simple-content-1 iko simple-content-2</code>
	<code>==&gt; tm:subclass-of ( tm:subclass : simple-content-1 , tm:superclass : simple-content-2 )</code>
[AB] <a href="#">path-expression</a>	<code>::= simple-content-1 isa simple-content-2</code>
	<code>==&gt; tm:type-instance ( tm:instance : simple-content-1 , tm:type : simple-content-2 )</code>
[55] <a href="#">postfix</a>	<code>::= filter-postfix   projection-postfix</code>
[54] <a href="#">postfixed-expression</a>	<code>::= ( tuple-expression   simple-content ) { postfix }</code>
[50] <a href="#">pragma</a>	<code>::= %pragma identifier QIRI</code>
[58] <a href="#">predicate-invocation</a>	<code>::= anchor ( &lt; anchor : value-expression &gt; [ , ... ] )</code>
[15] <a href="#">prefix</a>	<code>::= <math>\wline+!</math></code>
[49] <a href="#">prefix-directive</a>	<code>::= %prefix identifier QIRI</code>
[35] <a href="#">prefix-operator</a>	<code>::= ...any in the predefined environment...</code>
[57] <a href="#">projection-postfix</a>	<code>::= tuple-expression</code>
[46] <a href="#">query-expression</a>	<code>::= [ environment-clause ]</code>
	<code>( select-expression   flwr-expression   path-expression )</code>
[51] <a href="#">select-expression</a>	<code>::= select &lt; value-expression &gt;</code>
	<code>[ from value-expression ]</code>
	<code>[ where boolean-expression ]</code>
	<code>[ order by &lt; value-expression &gt; ]</code>
	<code>[ unique ]</code>
	<code>[ offset value-expression ]</code>
	<code>[ limit value-expression ]</code>
[21] <a href="#">simple-content</a>	<code>::= anchor [ navigation ]</code>
[18] <a href="#">step</a>	<code>::= ( &gt;&gt;   &lt;&lt; ) axis [ anchor ]</code>
[A] <a href="#">step</a>	<code>::= &gt;&gt; instances</code>

	$\Rightarrow$	$\Leftarrow$	<b>types</b>
[B] <a href="#">step</a>	::=	$\gg$	<b>subtypes</b>
	$\Rightarrow$	$\Leftarrow$	<b>supertypes</b>
[C] <a href="#">step</a>	::=	$\triangleright$	<a href="#">anchor</a>
	$\Rightarrow$	$\gg$	<b>players</b> <a href="#">anchor</a>
[D] <a href="#">step</a>	::=	$\Leftarrow$	<a href="#">anchor</a>
	$\Rightarrow$	$\Leftarrow$	<b>players</b> <a href="#">anchor</a>
[E] <a href="#">step</a>	::=	$\Leftarrow$	<a href="#">anchor</a>
	$\Rightarrow$	$\gg$	<b>traverse</b> <a href="#">anchor</a>
[F] <a href="#">step</a>	::=	@	
	$\Rightarrow$	$\gg$	<b>scope</b>
[G] <a href="#">step</a>	::=	=	
	$\Rightarrow$	$\Leftarrow$	<b>locators</b>
[H] <a href="#">step</a>	::=	$\approx$	
	$\Rightarrow$	$\Leftarrow$	<b>indicators</b>
[I] <a href="#">step</a>	::=	!	
	$\Rightarrow$	$\gg$	<b>item</b>
[J] <a href="#">step</a>	::=	$\approx$	
	$\Rightarrow$	$\gg$	<b>reifier</b>
[K] <a href="#">step</a>	::=	$\Leftarrow$	
	$\Rightarrow$	$\Leftarrow$	<b>reifier</b>
[11] <a href="#">string</a>	::=	/"([^\"] \\")*/   /'([^']* \\')*/	
[32] <a href="#">tm-content</a>	::=	"" <a href="#">ctm-instance</a> ""	
[24] <a href="#">tuple-expression</a>	::=	( < <a href="#">value-expression</a> [ <b>asc</b>   <b>desc</b> ] > )	
[P] <a href="#">tuple-expression</a>	::=	<b>null</b>	
	$\Rightarrow$	( )	
[3] <a href="#">undefined</a>	::=	<b>undef</b>	
[33] <a href="#">value-expression</a>	::=	<a href="#">value-expression</a> <a href="#">infix-operator</a> <a href="#">value-expression</a>   <a href="#">prefix-operator</a> <a href="#">value-expression</a>   <a href="#">function-invocation</a>   <a href="#">content</a>	
[43] <a href="#">variable</a>	::=	/[\$@%][\w#]+!*/	
[V] <a href="#">variable</a>	::=	:	
	$\Rightarrow$	<b>\$0</b>	
[44] <a href="#">variable-assignment</a>	::=	<a href="#">variable</a> <b>in</b> <a href="#">content</a>	
[28] <a href="#">xml-attribute</a>	::=	[ <a href="#">prefix</a> ] <a href="#">xml-fragments</a> $\equiv$ " <a href="#">xml-fragments</a> "	
[25] <a href="#">xml-content</a>	::=	{ <a href="#">xml-element</a> }	
[26] <a href="#">xml-element</a>	::=	$\leq$ <a href="#">xml-tag</a> { <a href="#">xml-attribute</a> } <a href="#">xml-rest</a>	
[30] <a href="#">xml-fragments</a>	::=	{ <a href="#">xml-text</a>   { <a href="#">query-expression</a> } }	
[29] <a href="#">xml-rest</a>	::=	$\triangleright$ $\geq$ { <a href="#">xml-element</a>   <a href="#">xml-fragments</a> } $\triangleleft$ <a href="#">xml-tag</a> $\geq$	
[27] <a href="#">xml-tag</a>	::=	[ <a href="#">prefix</a> ] <a href="#">xml-fragments</a>	
[31] <a href="#">xml-text</a>	::=	...see text...	

## Bibliography

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

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