



Topic Maps Query Language 0.11

Washington DC, November 14, 2004

Tutorial

<http://www.isotopicmaps.org>

slide 1

So, what are we doing?

- **Tutorial in TMQL as it currently stands**
- **We have a working draft describing a TMQL**
 - we just don't think it's ready for publication
- **The Tokyo meeting cleared up a number of things**
 - the editors now understand each other sufficiently to be able to teach the language (which is progress :-)

TMQL – the big picture

- **TMQL has three kinds of queries**
 - path expressions, which return single values, sets, or sequences
 - select queries, which return tables, and
 - FLWR (flower) queries, which return many kinds of things
- **Select and FLWR queries can use path expressions**

Accuracy

- I *think* I have understood Robert's parts of this correctly
- However, I don't *know* that for certain
- So, what I'm presenting here is my view of this, somewhat restated from Robert's terms

Path expressions



*Common sublanguage
Simple expressions*

Path expressions

- **XPath-like common sublanguage for all TMQL parts**
- **Works like this:**
 - the first subexpression produces a set/sequence of values,
 - after this come the steps, which apply operations to the set/sequence to produce a new one
 - the steps are chained in a sequence
 - the end result is the set/sequence produced by the last step

Typical first step

- **\$m**
 - reference to variable 'm' sent as a parameter to the query
 - the result of this is the topic map itself (that is, the node representing it)

Robert likes this approach; personally I'd prefer to avoid the variable, and instead have the topic map be implicit

Typical second step

- **\$m / composer**
 - the '/' produces all topics and associations in the topic map, then filters them according to the value produced by the 'composer' expression
 - the 'composer' expression evaluates to the composer topic
 - the filtering is by type, so the result is all topics of type 'composer' (or some subtype thereof)

Robert and I *both* like this bit :-)

Three typical steps

- **\$m / composer # date-of-birth**
 - the '#' produces all topic names and occurrences of the topics produced by '\$m / composer', then filters them by type so that only the date-of-birth occurrences remain
- **\$t # date-of-birth**
 - same as above, but starting from a variable '\$t' containing a topic
- **puccini # date-of-birth**
 - would find Puccini's date of birth
 - Robert claims this isn't allowed; LMG not sure document says that, or even that it should

LMG comments on the '#' operator

- **Robert has defined '#' as an expansion to syntax that operates on a 'virtual association' between topics and base name/occurrence**
- **He has two reasons for this**
 - one is how he *appears* to have modelled TMDM using Tau,
 - the other is that he thinks having different operators for conceptually different operations in large path expressions make them easier to read
- **Personally, I do *not* like this**
 - I feel the operation is the same, and that defining all steps as '/' would be much cleaner
 - I also don't feel the underlying metamodel should be exposed directly

Less typical steps

- **\$t -> composer \ composed-by / work**
 - '-> composer' selects the associations in which the topic '\$t' plays roles of type 'composer'
 - '\ composed-by' filters out those associations which are of type 'composed-by'
 - '/ work' produces all association roles in these associations, then filters them by type so only the 'work' roles remain, then selects the topic playing those roles
 - in short, the works composed by the topic(s) in '\$t'

Not very happy with the '->' operator being different from the '/' operator. Robert's arguments are readability, and also that it's doing something different

Filtering with “predicates”

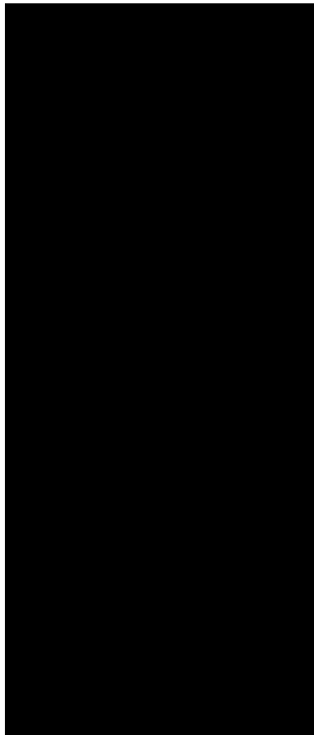
- **\$m / opera [premiere-date < “1900-01-01”]**
 - the '[...]' is evaluated relative to the value(s) produced by the expression before it, and filters out everything for which the expression within it is not true
 - in short, this is all operas premiered before 1900
 - predicates can be applied to any step

The term “predicate” is (unfortunately) used to mean both “tolog predicate” and “XPath predicate”, and these are completely different

Dealing with scope

- **\$t / @ english**
 - this will produce any characteristics in the English scope
- **\$t # bn @ english**
 - this will only produce base names in the English scope
- **\$t # oc @ english**
 - only external occurrences
- **\$t # rd @ english**
 - only internal occurrences

Select queries



More complex queries

Basic form

- **Select queries take the following form (blue parts being optional)**

select ...

from ...

where ...

order by ...

unique

?

Non optional part!

A thin black arrow points from the text 'Non optional part!' to the 'where ...' line of the SQL query.

Predicates

- **Predicates here take the form**
predicate-name (parameter1 , parameter2)
- **Parameters can either be literals or variables (\$variable)**
 - Literals constrain the result
 - Variables produce results (unless, of course, they are bound already, in which case they also constrain)

A simple example

- **instance-of(\$A, composer)?**
 - finds all instances of the 'composer' type (and its subclasses)
 - these are bound to the variable \$A
 - the result is returned as a single-column table with one row per composer
- **note that the real syntax is as follows:**
 - \$A : composer?

Treating association types as predicates

- **composed-by(puccini : composer, \$O : work)?**
 - find all \$Os which have a composed-by association with 'puccini'
- **composed-by(\$C : composer, \$O : work)?**
 - find all composer/work pairs
- **composed-by(\$C : composer, toscia : work)?**
 - find the composer(s) of the work “tosca”
- **composed-by(puccini : composer, toscia : work)?**
 - is it true that Puccini composed Tosca?

Treating occurrence types as predicates

- **\$WORK : opera, premiere-date(\$WORK, \$DATE), \$DATE < “1900-01-01”?**
 - finds first all work/date-combinations, then filters by date
 - note that this also demonstrates chaining of predicates
- **\$WORK : opera, \$WORK / premiere-date < “1900-01-01”?**

Expressing alternatives

- **\$OPERA : opera, {
 composed-by(\$OPERA : work, puccini : composer) |
 composed-by(\$OPERA : work, verdi : composer)
 }?**
 - finds all operas composed by Puccini *or* Verdi
 - each branch can contain full predicate lists

Optional clauses

- **\$OPERA : work, { premiere-date(\$OPERA, \$DATE) }?**
 - finds all operas and their premiere dates *if they have one*
 - the optional clause can contain any form of predicate list
- **select \$OPERA, \$OPERA / premiere-date where \$OPERA : work?**
 - alternative solution using path expressions

Not sure we need this any more. Leaving it in for the time being.

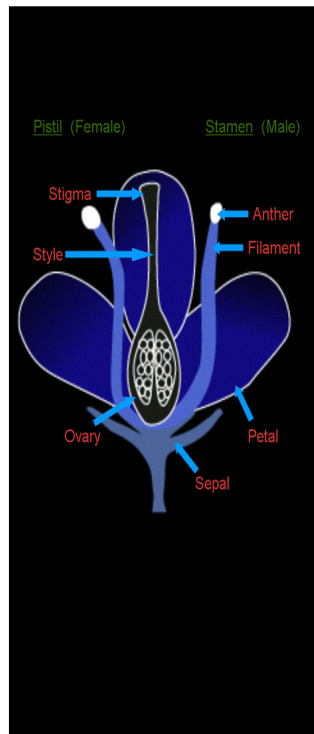
Expressing negation

- **born-in(\$PERSON : person, \$CITY : place),
not(located-in(\$CITY : container, italy : containee))?**
 - not can contain any form of predicate list

Non-existential queries

- Normal parts of select queries match so long as it is true that there exists *something* which matches the query
- We also want to be able to say that we want to find things where *every candidate* meets some particular condition
- **select \$TEAM where**
 \$TEAM : team,
 every team-member(\$TEAM : team, \$PLAYER : member)
 satisfies is-injured(\$PLAYER : patient)?
- **Can also be solved differently**
 - \$TEAM : team,
 not(team-member(\$TEAM : team, \$PLAYER : member),
 not(is-injured(\$PLAYER : patient)))?
 - that is, find all teams in which there is not (a team member who is not (injured))

FLWR queries



Even more complex queries

Some background

- **FLWR queries are syntactically inspired by XQuery**
- **The heart of them is predicate lists, like with select queries**
 - however, the predicate list syntax is different
 - it's different because Robert didn't like the select syntax, and I didn't like his
 - so feedback on which is the better syntax would be welcome

Basic structure

- **The structure of FLWR queries is (optional bits in blue)**

for \$foo in ..., \$bar in ...

for \$foo2 in ..., \$bar2 in ...

let \$baz := ...

let \$qux := ...

where ...

return ...

order by ...

unique

RETURN

- **return (puccini, puccini # date-of-birth, puccini # date-of-death)**
 - creates a 3-tuple consisting of the values produced by the path expressions
 - this is the result of the query
- **In general, RETURN produces the query result**
 - this can be through projection, like in select expressions
 - it can also be generation of XML content or TM results
 - the last two not covered by the existing draft

FOR

- **FOR** creates a loop over the **sequence/set** of results produced by the expression after **IN**
- **FOR \$composer IN \$m / composer**
RETURN (\$composer # bn, \$composer # date-of-birth)
 - returns a sequence of 2-tuples, one for each composer

FOR (2)

- **FOR \$composer IN \$m / composer**
FOR \$opera IN \$composer -> composer \ composed-by / work
RETURN (\$composer # bn, \$opera # bn)
 - returns all composer name, opera name pairs

WHERE

- **FOR \$composer IN \$m / composer**
WHERE composed-by(\$composer : composer, \$opera : work)
RETURN (\$composer # bn, \$opera # bn)
 - identical to previous query

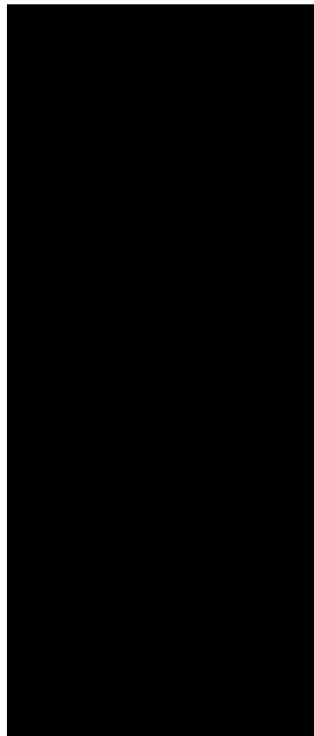


LET

Longer WHERE

- **WHERE**
composed-by(\$OPERA : work, \$COMPOSER : composer) AND
based-on(\$OPERA : result, \$WORK : source) AND
written-by(\$WORK : work, \$AUTHOR : author)
RETURN
(\$COMPOSER # bn, \$AUTHOR # bn)

Declarations



Common to all sub-languages

Declarations

- **URI prefix declarations**
- **Import declarations**
- **Rule declarations**
- **Function declarations**