The NITE Query Language

Stefan Evert and Holger Voormann

Version 2.1
26 March 2002

Contents

1 Introduction and overview

The **NITE query language** (NiteQL) specified in this document is based on the MATE query language[1] (Q4M). Some additions were necessary, and the semantics of query evaluation were defined more clearly and accurately. The support for structural relations was completely re-designed on the basis of the formal specification of the **NITE object model** (NOM), and the time comparison operators were extended with optional rules for "fuzzy" matching. Complex queries and quantifiers with variable scope make NiteQL more powerful and expressive than its predecessor. Additionally, certain syntactic details were changed to make parsing with a `lex`/`yacc`-style grammar easier and more robust[2] as well as for the sake of better compatibility with XPath and other W3C standards.

Is is important to keep in mind that the NITE query language operates on the level of the NITE object model and was designed for maximal expressiveness and flexibility. Thus, its view of a NOM corpus will not always coincide with the interpretation of the same data by a user. For instance, an ordered tree could represent the syntactic analysis of a sentence or a hierarchy of annotation codes, among many other possibilities. Although the two kinds of trees are entirely dissimilar from the perspective of a user, they are structurally isomorphic and neither the object model nor the query language make a distinction between them. Another example is the distinction between annotation data and meta-data (information about the recordings, transcriptions, and annotations that make up a multi-modal corpus): both can be searched with the same query syntax although the interpretation of the search results will be fundamentally different. The interpretation of the data structures that make up a NOM corpus is the topic of the **NITE data model**. It is envisaged that a pre-processing or template mechanism will later allow users to write queries in a simpler, but less expressive language that operates in terms of this abstract data model. Internally, such queries will be translated into equivalent NiteQL expressions for evaluation by the query processor.

The basic structure of a NiteQL query is similar to that of SQL and XQuery. First, *variables* representing corpus elements are declared (and can be bound to a particular element type). Then, these variables are bound to suitable elements so that the *match condition* – a Boolean expression over attribute tests, structural constraints, and time constraints – is satisfied. The result of a *simple NiteQL query* is a list of n-tuples of elements, representing matching variable bindings. Variable can be *bound internally* with existential and universal quantifiers, in which case they will not be part of the query result. Finally, simple queries can be combined into *complex queries*, giving hierarchically structured results.

A query engine implementation may choose to support only a subset of the full NiteQL specification. The core features, which all implementations must provide, are simple queries with all attribute tests, structural and temporal relations, as well as the `exists` quantifier. Advanced implementations will also support complex queries, scopes (for local variables and quantifiers), and the `forall` quantifier. Further useful extensions are a template mechanism and/or user-defined operators and functions, which could also be used to provide syntactic sugar for the non-expert user.

Several possibilities for the implementation of a NiteQL query engine have been considered: (i) The translation into an XQuery expression would be fairly straightforward. Unfortunately, the XQuery standard is just being finalised and there are no free and efficient implementations

[1]see `http://www.ims.uni-stuttgart.de/projekte/mate/WB3/Q4M/001/docu/quer.html`

[2]In particular, attribute references like `$w orth` were changed to `$w@orth`, which avoids ambiguities in connection with reserved keywords and (pre-defined as well as user-defined) lexical operators auch as `and`, `or`, `not`, and `overlaps.with`.

available at the time of writing. (ii) It is theoretically possible to compile a NiteQL query into a very large and complex XSLT stylesheet. However, there are many practical difficulties, especially when the full scope of the query language is supported, and the resulting stylesheets are likely to be too slow for sizeable corpora.[3] It is still a tempting solution for queries that are embedded into stylesheets. (iii) The best solution seems to be a custom implementation (written in Java), perhaps using standard XML and XPath libraries to facilitate attribute tests and object tree navigation. This is the recommended implementation strategy. (iv) When the corpus is stored in a relational database rather than a set of XML files, it should be possible to compile a NiteQL query into an equivalent SQL query. This will require special provisions in the database design, though, especially a full representation of the dominance and precedence relations.

This document assumes that the reader is familiar with the concepts and terminology introduced in the **NITE Object Model** specification. The syntax of the NITE query language is presented in a variant of EBNF (ISO/IEC 14977: 1996). For brevity, non-terminal symbols are shown in *italics* and literal symbols in `typewriter` font (so that special quotes and colons can be omitted). Multiple instances of the same non-terminal in a grammar rule may be distinguished by subscripted numbers for reference in the main text. Complex terminal symbols (that are identified by a lexical analyser) are not defined in the EBNF notation, but described informally in the main text. Alternatives are separated by a vertical bar |, parentheses (. . .) are used for grouping, square brackets [. . .] indicate optionality, and curly braces {. . .} indicate repetition (zero-or-more). Some care has to be taken to distinguish these meta-characters from the corresponding literals (`([{|}])`).

2 General structure of a query

A **simple NiteQL query** consists of a **variable declaration** part and a **match condition**. Variables represent corpus elements (usually of a certain type), and the match condition is a Boolean expression over attribute tests, structural relations, and temporal relations between elements. The **result** of a query is a list of **variable bindings** (mapping each variable to a specific element in the corpus) which satisfy the match condition. It can also be interpreted as a list of n-tuples of elements (where n is the number of declared variables that are not internally bound, cf. Section 5). The declarations part is separated from the match condition by a `:` character.

simple query =
  declarations `:` *match condition*
  | *declarations* [`:`]

The *match condition* may be empty, in which case it always evaluates to true and the `:` delimiter is optional. A **complex query** consists of a sequence of single queries separated by `::` markers. Each query in the sequence operates on the "fixed" result of the previous query (see Section 4 for a more detailed explanation).

complex query =
  simple query { `::` *simple query* }

[3]This assessment is supported – and exacerbated – by our observation that current XSLT processor seem to be optimised for the simple structural transformations required by document formatting applications.

In the **variable declaration** part, variables representing corpus elements are declared. Usually, each variable is assigned to a certain element type (i.e. it may only be bound to elements of this type). It is also possible to give a list of alternative element types or to leave the element type unspecified.

declarations =
  variable declaration { *variable declaration* }

variable declaration =
  `(` *var* *type* `)`
  | `(` *var* { `,` *var* } *type* { `|` *type* } `)`
  | `(` *var* { `,` *var* } `)`

The *type* must be one of the element types defined for the corpus. A variable *var* consists of a `$` character followed by the variable name, e.g. `$w`, `$phone1`, and `$$` (for the context element within a stylesheet). It is also possible to declare several variables in a single statement.

NB: Different variables (with compatible type declarations) may be bound to the same element, unless they are explicitly required to be different (cf. Section 3.2). It is therefore easy to get swamped with unwanted, but formally correct matches by overlooking the possibility of identical bindings. For instance,

```
($w1 word) ($w2 word) ($s s):
  ($w1@orth == "I") && ($w2@orth == "I") && ($s ^ $w1) && ($s ^ $w2)
```

will return every occurrence of the word *I*, with `$w1` and `$w2` bound to the same `word` element. The intended result is obtained by adding the condition `($w1 != $w2)`.

Comments are allowed in the form of line comments and block comments. **Line comments** start with the symbol `//` and extend to the end of the current line. **Block comments** are delimited by `/*` and `*/`, and may extend over multiple lines. Both types of comments are removed before query evaluation.

3 Match condition

The **match condition** is a Boolean expression over attribute tests as well as structural and temporal relations between elements. Boolean operators can either be written as the C-style symbols `!` (*not*), `&&` (*and*), `||` (*or*) used by Q4M, or as the more readable XPath versions `not`, `and`, `or`. Parentheses are used for grouping. The operators are listed in the order of their precedence below.

match condition =
  `(` *match condition* `)`
  | `!` *match condition*
  | `not` *match condition*
  | *match condition* `&&` *match condition*
  | *match condition* `and` *match condition*
  | *match condition* `||` *match condition*
  | *match condition* `or` *match condition*
  | *match condition* `->` *match condition*

| *attribute test*
| *structural relation*
| *time relation*

The implication operator mc_1 `->` mc_2 is logically equivalent to the expression mc_2 `|| !(`mc_1`)`. It is mainly used in conjunction with the `forall` quantifier (see Section 5). Every match condition can be translated into an equivalent disjunctive **normal form** defined by the grammar below.

match condition =
term { `||` *term* }

term =
factor { `&&` *factor* }

factor =
[`!`] `(` *attribute test* `)`
| [`!`] `(` *structural relation* `)`
| [`!`] `(` *time relation* `)`

In most implementations, query evaluation will be based on this normal form, and optimisation strategies will usually involve a reordering of the factors in each term. The query engine should evaluate "cheap" conditions first to establish some variable bindings, and then follow relative paths (i.e. structural or temporal relations) to elements specified by more "expensive" conditions. Negated factors will typically be evaluated as late as possible because they are likely to match a large number of elements. Note that this reordering strategy rules out any short-circuit evaluation semantics for Boolean operators. Special optimisations may be needed for the `->` operator. Especially when used as the top-level operator within a scope, its translation into the equivalent normal form should be avoided.

3.1 Attribute tests

An **attribute test** compares the attribute value of an element to a constant (either a number or a string) or to another attribute value (of the same element or a different one). The precise semantics of attribute tests are implementation dependent, and are not defined in this document. In particular, implementations may or may not make a difference between numerical, categorical, and string-valued attributes. It is recommended to follow the XPath standard wherever possible. In particular, NiteQL should be **weakly typed** so that strings can be converted to numbers when necessary (i.e. when a comparison operator or other computation expects a numerical operand), and vice versa. This is consistent both with the NITE object model and the XML standard, where all attribute values are strings *per se* (but may be *interpreted* as floating-point numbers). No distinction should be made between integer and floating-point numbers. (Note that an implementation of the NiteQL language may override these recommendations for optimisation purposes.)

attribute test =
expr relop expr
| *expr*

relop =
  `==` | `eq`
  | `!=` | `ne`
  | `>=` | `ge`
  | `<=` | `le`
  | `>` | `gt`
  | `<` | `lt`

The last form of an attribute test, which consists of a single *expr*, is evaluated to true or false according to the XPath rules. Note that the operators `>=`, `<=`, `>`, `<` can also be used to compare strings according to the collating sequence of the current locale. The more readable lexical operators `eq`, `ne`, `ge`, `le`, `gt`, and `lt` are particularly useful for queries embedded in stylesheets.

expr =
  string constant
  | *number constant*
  | *attribute value*

Expr is either an attribute value or a constant, where **constants** (both numbers and strings) are defined as in XPath. In particular, string constants must be enclosed either in single quotes (`'...'`) or in double quotes (`"..."`). An **attribute value** is accessed through a variable bound to the relevant element and the name of the attribute, separated by a `@` character (again similar to the XPath notation). A NiteQL implementation may extend the *expr* rule to cover complex expressions calculated from attribute values and constants. These calculations may involve, for instance, the elementary arithmetic and string manipulation functionality of XPath. However, for optimisation reasons, one side of each attribute test should *always* be a simple attribute value.

attribute value =
  var`@`*attribute*
  | `TEXT(` *var* `)`
  | `ID(` *var* `)`

Remember that *var* starts with a `$` character. *attribute* must be the name of an attribute defined for elements of the respective type (as indicated by the variable declaration). The built-in function `TEXT()` corresponds to the TEXT() operator of the NITE object model and returns the textual content of an element. For an XML-based implementation, `ID()` returns the value of the special `nite:id` attribute, which should be a unique identifier of the respective element. Examples of attribute value references are `$word@orth` and `$turn@speaker`, as well as `TEXT($np)` for the textual content of the element `$np`. Note that attribute tests involving the `TEXT()` function are more "expensive" than simple attribute references (of the form *var*`@`*attribute*).

In addition to numerical or string comparison, an attribute value can be matched against a **regular expression**. In this case, the LHS of the comparison must be a simple attribute value (including the `TEXT()` function), and the RHS must be a string constant which is interpreted as a POSIX regular expression. The regular expression must match the entire attribute value rather than just a substring (in contrast to the `grep` program and similar applications).

attribute test =

  attribute value ~ *string constant*
  | *attribute value* `!~` *string constant*

The start and end times of **timed elements** (corresponding to the special $f_{\texttt{start}}$ and $f_{\texttt{end}}$ attributes in the NITE object model) can be accessed through built-in `START()` and `END()` functions, which return floating-point values. Additional functions are available to obtain the length (`DURATION()`) and midpoint (`CENTER()`) of the time interval associated with an element. If *var* is not bound to a timed element, the entire attribute test containing *expr* evaluates to false. Timed elements can be identified with the `TIMED()` function:

expr =

  `START(` *var* `)`
  | `END(` *var* `)`
  | `DURATION(` *var* `)`
  | `CENTER(` *var* `)`
  | `TIMED(` *var* `)`

The query engine should reorder attribute tests (within each term of a normal form) and evaluate the easy ones first. An attribute test is *easy* if the LHS is a simple attribute value, and the RHS is a constant. Matching elements can then be looked up quickly, provided that an index has been built. The *less easy* attribute tests involve regular expressions and the built-in functions `TEXT()`, `DURATION()`, etc. Everything else is *really hard*, especially when the implementation allows complex expressions (with arithmetic or string manipulation functions).

The reason why we need built-in functions for the start and end times of elements (rather than accessing the attributes directly) is that an implementation based on the standard XML encoding will store the special time attributes in the NITE namespace. Thus `$w@start` refers to a user-defined attribute with the name `start`, while `START($w)` refers to the special $f_{\texttt{start}}$ attribute (named `nite:start` in the standard encoding).

Note that the XPath comparison operator `=` is renamed to `==`, for the sake of consistency with other C-style operators. Implementations are free to add further synonyms for operators, such as `=`, `&`, `|` for `==`, `&&`, and `||`.

3.2 Structural relations

This section describes operators that give access to the explicit and implicit structural information summarised in the NOM specification. It is assumed that the reader is familiar with this summary, and the following definitions use the terminology introduced there. Please refer to the NITE Object Model document for the precise semantics of structural relations.

The grammar rules below describe relations between two or three elements, represented by indexed variables var_i. In the main text, x_i stands for the corpus element to which the variable var_i is bound. H stands for a named hierarchy or hierarchy collection and L for a (linear) layer that appear in some of the rules.

structural relation =

  var `==` *var*
  | *var* `!=` *var*

The simplest structural relation asserts the identity or non-identity of two elements. Since different variables may be bound to the same element, the `!=` operator is often necessary to exclude unwanted results. The `==` operator is less useful and was mainly added for the sake of symmetry.

structural relation $=$

var$_1$ `^` *var*$_2$
| *var*$_1$ `^`*v* *var*$_2$
| *var*$_1$ `^[`*v*`][`*n*`]` *var*$_2$

The first form of the **dominance operator** `^` represents the dominance relation $x_1 \Uparrow x_2$. More precisely, it is defined as the closure of direct dominance $x_1 \uparrow^* x_2$, so that `$x ^ $x` is always true. The second form specifies an exact vertical distance between x_1 and x_2, i.e. the length of the path from x_1 to x_2. Note that $|P(x_1, x_2)| = v + 1$ and `^1` corresponds to direct dominance (where $|P(x_1, x_2)| = |\{x_1, x_2\}| = 2$). The third form refers to the horizontal axis formed by the elements at a given vertical distance v from x_1 in the offspring tree $O(x_1)$, to which x_2 must belong. n is the position of x_2 in this axis with respect to the axial ordering. If v is omitted, the horizontal axis is the boundary of the offspring tree $O(x_1)$, and x_2 must be a leaf in $O(x_1)$. A negative value of n starts counting from the end of the axis. For instance, `$p ^1[1] $c` matches the first child of `$p`, and `$p ^1[-1] $c` its last child).

structural relation $=$

var$_1$ `<`[*named hierarchy* | *var*$_3$]`>` *var*$_2$
| *var*$_1$ `<`[*named hierarchy* | *var*$_3$]`>`*h* *var*$_2$

The **precedence operator** `<>` requires that x_1 and x_2 belong to the named hierarchy H (or hierarchy collection $\{H_i\}$) identified by *hierarchy*, and that $x_1 \prec_H x_2$. Note that `$x <> $x` is *never* true. The hierarchy can also be specified in the form of an offspring tree $O(x_3)$ (where *var*$_3$ indicates the root element x_3), equivalent to the condition $x_1 \prec_{O(x_3)} x_2$. Note that `$x <$a> $y` implies `$a ^ $x` and `$a ^ $y`, since x_1 and x_2 must belong to the tree $O(x_3)$ (i.e. $x_3 \Uparrow x_1 \wedge x_3 \Uparrow x_2$). If the hierarchy is unspecified, precedence is interpreted wrt. some offspring tree $O(z)$, corresponding to the condition $\exists z \in E : z \Uparrow x_1 \wedge z \Uparrow x_2 \wedge x_1 \prec_{O(z)} x_2$. The second form additionally requires an exact sequential distance h between x_1 and x_2 with respect to the given hierarchy or offspring tree.

It is interesting to note that the relation `$x <> $y` can also be expressed with an explicit `exists` quantifier, using the scope notation introduced in Section 5: `{(exists $w): $x <$w> $y}`. A NiteQL implementation might make this substitution internally before evaluating the query.

structural relation $=$

var$_1$ *v*`<`*var*$_3$`>`*h* *var*$_2$
| *var*$_1$ *v*`<`[*named hierarchy*]`>`*h* *var*$_2$

This form of the precedence operator refers to the distance between x_1 and x_2 on a horizontal axis (i.e. the number of intervening elements according to the axial ordering plus one). The axis consists of all elements at a given vertical distance v from x_3 in the offspring tree $O(x_3)$. A vertical distance of $v = -1$ selects the boundary of $O(x)$. Of course, x_1 and x_2 must belong to the respective horizontal axis. The second form stipulates that some element x_3 matching these conditions exists, which must also satisfy $x_3 \in H$ if a named hierarchy H is specified.

Particularly useful forms of the precedence operator are `<>1` for immediate neighbours (especially wrt. a named hierarchy, e.g. `<orth>1` for orthographically adjacent words), and `1<>1` for adjacent siblings.

expr =
  `D(`*layer*`,` *var*$_1$`,` *var*$_2$`)`
  | `DA(`*layer*`,` *var*$_1$`,` *var*$_2$`)`

The **horizontal distance** represented by the built-in functiond `D()` and `DA()` is computed by projection to a linear layer (this procedure is detailed in the NOM specification). The first form requires $x_1 \prec x_2$ and returns the horizontal distance between x_1 and x_2 obtained by projection to the layer L. If the distance cannot be determined (e.g. the projections $P_L(x_1)$ and $P_L(x_2)$ fall into different components of L or one of them is empty), the entire attribute test containing the *expr* evaluates to false. The second form (**absolute horizontal distance** `DA(L,$x,$y)`) is valid when either `D(L,$x,$y)` or `D(L,$y,$x)` is defined and returns the respective value.

structural relation =
  var$_1$ `>`[*role*] *var*$_2$
  | *var*$_1$ `>`[*role*]`^` *var*$_2$

Another structural relation is the **pointer** operator `>`. The first form matches a pointer from x_1 to x_2 labelled with the role r (i.e. requires that $x_1 \rightarrow_r x_2$). The *role* is specified as a string value enclosed in single or double quotes. (Recall that an element may have multiple pointers with the same role.) If *role* is omitted, the relation matches any pointer regardless of its role, corresponding to the condition $x_1 \rightarrow x_2$. The **pointer-to-subtree** operator `>^` combines `>` and `^`, and is especially useful in conjunction with hierarchies of annotation codes. It evaluates to true iff there is an element $w \in E$ such that $x_1 \rightarrow_r w$ and $x_2 \uparrow^* w$.

Of course, w is not part of the query result. As in the case of the `<>` operator, the pointer-to-subtree relation can also be expressed through a scoped quantifier: `{(exists $w): $x > $w && $y ^ $w}` is equivalent to `$x >^ $y`.

3.3 Temporal relations and fuzziness

The **time operators** of Q4M were supplemented with the fundamental inequalities

  `starts.earlier.than`, `starts.later.than`,
  `ends.earlier.than`, and `ends.later.than`,

from which most other operators can be constructed. Since the various symbols introduced for time operators always were a source of confusion, use of the more readable *lexical operators* listed in the second column of the table below is recommended.

time relation =
  var$_1$ *op*[*modifier*] *var*$_2$

where *op* is one of the operators from the following list (s_1, e_1 are the start and end times of the binding x_1 of *var*$_1$, and s_2, e_2 are those of the binding x_2 of *var*$_2$).

op	lexical operator	description	definition
`%`	`overlaps.left`	left overlap	$s_1 \leq s_2 \wedge e_1 \geq s_2\, (\wedge\, e_1 \leq e_2)$
`[[`	`left.aligned.with`	left alignment	$s_1 = s_2$
`]]`	`right.aligned.with`	right aligment	$e_1 = e_2$
`@`	`includes`	inclusion	$s_1 \leq s_2 \wedge e_1 \geq e_2$
`[]`	`same.duration.as`	identical duration	$s_1 = s_2 \wedge e_1 = e_2$
`#`	`overlaps.with`	overlap	$\neg(e_1 \leq s_2 \vee e_2 \leq s_1)$
`][`	`contact.with`	contact	$e_1 = s_2$
`<<`	`precedes`	temporal precedence	$e_1 \leq s_2$
	`starts.earlier.than`	earlier start time	$s_1 \leq s_2$
	`starts.later.than`	later start time	$s_1 \geq s_2$
	`ends.earlier.than`	earlier end time	$e_1 \leq e_2$
	`ends.later.than`	later end time	$e_1 \geq e_2$

Due to the nature of multi-modal / multi-media communication, time operators (especially alignment, identical duration and contact) cannot be interpreted as strict equalities. Therefore, points on the timeline that are within a certain **fuzziness interval** of each other must be considered equal. The exact size of this interval is implementation-dependent, and should be based on cognitive limits and user experiences. The size fuzziness interval should be configurable (e.g. in a corpus schema, or within a NiteQL-based stylesheet) and may depend on the type of signal to which annotations refer (e.g. video vs. audio recordings). Within a query, the amount of fuzziness can be controlled by a **modifier** appended to one of the time operators.

modifier =
`++` | `+` | `-` | `--`

By default (without modifier), the equalities and inequalities in the table above are matched exactly. The modifiers `+` and `++` enforce a stricter interpretation of the time relations, while `-` and `--` allow more lenience.

As noted above, all time operators can be broken down into combinations of fundamental equalities and inequalities for start and end times (given in the table above), and thus into attribute tests involving the `START()` and `END()` functions. For instance, inclusion `$a @ $b` is equivalent to `$a starts.earlier.than $b && $a ends.later.than $b`. An implementation of the query language should use a similar approach and provide optimisations for the fundamental time equalities and inequalities (e.g. by maintaining a sorted list of the start and end times of all elements).

It might be sufficient to define fuzziness intervals for the fundamental equalities and inequalities (although very short and very long element durations may have to be taken into account, e.g. `$a @ $b` may evaluate to true for short non-overlapping elements if a fixed fuzziness interval is used). Below is a suggestion for the interpretation of time equalities ($t_1 = t_2$) and inequalities ($t_1 < t_2$).

equality	modifier	interpretation
$t_1 = t_2$	`++`	$t_1 = t_2$
$t_1 = t_2$	`+`	$t_1 = t_2$
$t_1 = t_2$		$t_1 = t_2$
$t_1 = t_2$	`-`	$\|t_1 - t_2\| < \epsilon$
$t_1 = t_2$	`--`	$\|t_1 - t_2\| < \delta$

inequalities	modifier	interpretation
$t_1 \leq t_2$	`++`	$t_1 \leq t_2 - \delta$
$t_1 \leq t_2$	`+`	$t_1 \leq t_2 - \epsilon$
$t_1 \leq t_2$		$t_1 \leq t_2$
$t_1 \leq t_2$	`-`	$t_1 \leq t_2 + \epsilon$
$t_1 \leq t_2$	`--`	$t_1 \leq t_2 + \delta$

Here, ϵ stands for the cognitive limit, i.e. the smallest time difference that can be distinguished by a human observer, and δ is the largest interval within which two events (usually from different modalities) can reasonably be construed to happen at the same time. The exact values of ϵ and δ will have to be fine-tuned during implementation. At least for corpora based on video recordings, which are typically played back at a speed of approximately 25 frames per second, $\epsilon \approx \frac{1}{25}s$ seems a good starting point.

Note that some operators can be broken down into fundamental inequalities in more than one way. Care has to be taken that the fuzziness definitions for the fundamental inequalities do not lead to counterintuitive results. For instance, the definition of the overlap operator `#` involves logical negation, so it may be necessary to swap `++` with `--` and `+` with `-` when it is evaluated in terms of the fundamental inequalities. It may also turn out to be necessary to adjust fuzziness intervals according to element durations and the modalities involved. A possible solution for the latter involves multiple aligned timelines with different granularities. As in the case of video frames, ϵ should be set to the granularity of the coarsest timeline involved in a comparison.

3.4 Regularity

Experience from the MATE project shows that novice users will often formulate queries with "unconnected" variables. Such a query is the logical conjunction (or worse, disjunction) of two or more match conditions involving disjoint sets of variables. The query result is thus the Cartesian product of the each of the "disjoint" match conditions, producing a large number of unexpected matches.

The concept of regularity is intended as a safety measure, warning the user about a potentially erroneous query and the long waiting time that has to be expected. It can be defined as a reordering of the normal form so that the first factor in each term is an "easy" condition, and that each "new" variable within the term is related to "known" variables to a structural or temporal operator *before* it is used in an attribute test.

There are some examples of valid but irregular queries (e.g. coreference between elements by a common ID code), so it should always be possible for advanced users to turn off such warnings.

4 Query results

The result of a simple NiteQL query is a list of n-tuples of elements (or, more precisely, variable bindings) satisfying the match condition, where n is the number of variables declared without quantifiers.[4] The query result is returned in the form of an XML document (or, abstractely, a new tree structure adjoined to the corpus). Each query match corresponds to a `match` element, with pointers representing variable bindings and the variable name given by the pointer's role. An example result for a query involving variables `$w` and `$p` is

```
<matchlist size="2">
  <match n="1">
    <nite:pointer role="w" xlink:href="..."/>
    <nite:pointer role="p" xlink:href="..."/>
  </match>
  <match n="2">
    <nite:pointer role="w" xlink:href="..."/>
    <nite:pointer role="p" xlink:href="..."/>
  </match>
</matchlist>
```

[4]See Section 5 for details on quantifiers.

For a **complex query**, the leftmost simple query is evaluated first. Then, for every match, the following simple query is evaluated with the variable bindings from the first query fixed, etc. The fixed variable bindings may be used anywhere in the ensuing subqueries (without affecting the regularity assessment). This evaluation strategy produces a hierarchically structured query result, where each match of the leftmost simple query includes a matchlist for the second query. In the example

```
($w word): $w@orth ~ "S.*" :: ($p phone): $w ^ $p
```

the query result has the following structure:

```
<matchlist size="2">
  <match n="1">
    <nite:pointer role="w" xlink:href="..."/>
    <matchlist type="sub" size="2">
      <match n="1">
        <nite:pointer role="p" xlink:href="..."/>
      </match>
      <match n="2">
        <nite:pointer role="p" xlink:href="..."/>
      </match>
    </matchlist>
  </match>
  <match n="2">
    <nite:pointer role="w" xlink:href="..."/>
    <matchlist type="sub" size="1">
      <match n="1">
        <nite:pointer role="p" xlink:href="..."/>
      </match>
    </matchlist>
  </match>
</matchlist>
```

5 Quantifiers and scopes

There are many examples of queries which require auxiliary elements to express complex structural relations. From the user's perspective, such auxiliary elements should certainly not be part of the query result. Even worse, in some cases more than one element might match an auxiliary variable, and the query result is inflated with effectively duplicate matches that just differ in the auxiliary elements. The mathematical solution to this problem are the ∃ (existential) and ∀ (universal) **quantifiers**. Such quantifiers "bind" a variable within a logical expression, yielding an expression with fewer variables.

variable declaration =
    `(` `exists` *var* { `,` *var* } *type* { `|` *type* } `)`
    | `(` `exists` *var* { `,` *var* } `)`

In NiteQL, the existential quantifier (`exists`) can be added to a any variable declaration. After evaluating the query, all quantified variables are removed from the query result, then duplicate matches (with respect to the remaining variables) are combined into a single match.

The actual implementation should be more efficient than the strategy outlined above. When the factors in each term of the normal form are re-ordered appropriately, it will often be possible to search bindings for quantified along relative paths. Backtracking ensures that all matches are found but that not duplicates (where the non-quantified variables are bound to the same set of elements) are considered.

variable declaration =
  (`forall` *var* { `,` *var* } *type* { `|` *type* })
  | (`forall` *var* { `,` *var* })

The universal quantifier (`forall`) can also be added to any variable declaration. However, it is only useful in connection with the implication operator `->`, because otherwise *all* elements of the specified type(s) would have to satisfy the match condition. Care has to be taken that all attribute tests and structural relations involving a universally quantified variable are embedded in the RHS of an implication whose LHS specifies a *subset* of elements that must also satisfy the other conditions. Contrast the typical mistake of a novice user

```
($w word) (forall $m morph): ($w ^1 $m) && ($m@type == "stem")
```

(which requires that *all* `morph` elements must be lexical stems and children of a single `word` element) to the correct form

```
($w word) (forall $m morph): ($w ^1 $m) -> ($m@type == "stem")
```

which finds words without affixes (all children of type `morph` must be marked as stems). When existential and universal quantifiers are mixed, the result depends crucially on the particular order of the variable declarations. Such queries should only be written by users who are familiar with the basic notions of propositional logic.

The usage of quantifiers discussed up to now poses several problems. As has been said, the `forall` quantifier should usually apply to a limited scope, often on the RHS of an implication. Negation of the `exists` operator would be desirable for many applications, but again it is only meaningful when its scope can be limited to a subexpression of the match condition. Finally, query evaluation that is based on a transformation into normal form will often be very inefficient, especially for universally quantified variables. In order to solve these problems, simple subqueries can be embedded as a term in the match condition.

match condition =
  `{` *simple query* `}`

Such a *match condition* (which will often be embedded as a single term in a larger Boolean expression), consisting of a simple query enclosed in curly braces, is called a **scope**. The variables declared within the scope are local to the *simple query*, and must all be quantified (both existential and universal quantifiers are allowed and may be mixed). The match condition of the inner query may also refer to global variables and variables from enclosing scopes (collectively referred to as *external variables*). Since all local variables are quantified, the scope does not return a list of elements but rather a Boolean truth value. It evaluates to true iff the *simple query* matches for the current bindings of the external variables, and false otherwise.

The strongly recommended form for scopes declares just a single local variable, followed by a match condition with an implication operator at the top level, as shown in the following example.

```
($w word): ($w@pos == "NN") &&
  { (forall $m morph): ($w ^1 $m) -> ($m@type == "stem") }
```

As a term in a Boolean expression, a scope can be negated so that it evaluates to true iff the inner query does *not* match, and vice versa. Scopes can thus be used to express negated existential quantification, which is explicitly limited to the scope. For instance, to find non-terminal elements (`nt`) in a syntax tree that do not dominate open-class words (`word` elements), one could use the query below.

```
($x nt): !{ (exists $y word): ($x ^ $y) && ($y@class == "open") }
```

An implementation of the query processor should treat a scope as a complex subexpression that is not modified when the containing match condition is transformed into normal form. Each scope should also be evaluated as a single condition or "subquery". If the inner match condition has the recommended implicational form (with a simple condition on the LHS), only the RHS of the `->` operator should itself be normalised. It will often be possible for an optimised query implementation to evaluate scopes efficiently, especially when they have this special form .

6 XSLT stylesheet integration

NiteQL will be deeply embedded in the XML processing architecture of the NXT, replacing XPath in the `select` and `match` attributes of XSLT stylesheets. Although XPath can still be used (it is needed to access attribute values) and may sometimes be more concise and more efficient than NiteQL, selecting nodes with XPath requires an initimate knowledge of the technical details of the XML encoding used by the underlying NOM implementation.

The main difference between a stand-alone query and an embedded query is that the latter is supplied with a **context node** by the invoking XSLT template. This node must be a corpus element and is represented by the special `$$` variable in the query. Although the `$$` variable is implicitly defined and bound to the context element, it must still be declared when it is used (and specifying an element type might allow certain optimisations).

The **result** of an embedded query must be a list of single element matches, i.e. a flat list of elements. The `$$` variable is never returned in the query result. Every other variable in the query but one must be quantified. The query result is converted to a node set and passed back to the XSLT processor. Note that NiteQL queries cannot be used to select attribute values; here XPath has to be used. For instance, the XPath solution for printing the orthography and the start and end times of a given `word` element would be

```
<xsl:template match="word">  <!-- XPath is more efficient here -->
  <xsl:value-of select="@orth"/>
  <xsl:value-of select="@nite:start"/>
  <xsl:value-of select="@nite:end"/>
</xsl:template>
```

Turn to Section 7 for examples of NiteQL queries embedded in stylesheets.

When queries are used in stylesheets, optimisation is a crucial factor. Typically, the same query will be invoked many times for different context elements. Many queries will be used to locate a particular descendant or ancestor, or an element overlapping with the context element. The query optimisers should transform such expressions into relative paths starting from the context element. In the example

```
<xsl:apply-templates
     select="($$) ($n noise): ($n@type == 'cough') && ($$ ^ $n)"/>
```

it would probably be prohibitively slow to identify all noises of type "cough" first and then search the context element among their ancestors. It is more efficient to start from the context element and scan all its descendants for a noise of the desired type.

7 Examples

This section presents some examples of NiteQL queries with a short informal description of their (intended) result. It is hoped that these examples will help to clarify the basic ideas behind the query language, and provide a good starting point for further discussion.

7.1 Simple examples

Pairs of a `word` element `$w` and any descendant `$p` of type `phone`:

```
($w word) ($p phone): $w ^ $p
```

Pairs of a `word` element `$w` and any of its children (bound to `$child`):

```
($w word) ($child): $w ^1 $child
```

For each `word` element `$w`, return a list of all `phone` elements dominated by `$w`.

```
($w word) :: ($p phone): $w ^ $p
```

Elements are considered ancestors of themselves, so the following query would match every `word` element in the corpus.

```
($w word): $w ^ $w
```

The same happens when two different variables can be bound to the same element:

```
($w,$v word): $v ^ $w
```

In order to make `$v` a "true" ancestor of `$w`, their non-identity has to be explicitly required:

```
($w,$v word): ($v ^ $w)  &&  ($v != $w)
```

If we know that `phrase`s are *exactly* two levels above `morph`s, using `^2` may be considerably faster than the unrestricted `^` operator.[5]

```
($m1,$m2 morph) ($p phrase): ($p ^2 $m1) && ($p ^2 $m2)
```

[5]Theoretically, the query optimiser might be able work all that out by itself from the meta-information about the layer structure of a corpus, but that is an extremely difficult task to implement.

A regular expression example: find "words" like *errrm*, which contain a sequence of three or more *r*'s and which are marked as hesitations (through a `hesitation` parent).

```
($w word) (exists $h hesitation): ($h ^ $w) && ($w@orth ~ ".*rrr.*")
```

If `words` and `phones` are both timed, the start time of each `word` should coincide with the start time of its first `phone`. This query finds pairs of `word` and `phone` elements which violate this condition (we need to know that `phones` are *immediate* children of `words` in order to select the first `phone`).

```
($w word) ($p phone):
  ($w ^1[1] $p) && TIMED($w) && TIMED($p) && (START($w) != START($p))
```

7.2 Examples with quantifiers

Find two orthographically identical `words` that are "siblings" wrt. a `phrase` ancestor (i.e. within the same phrase). This query returns `word` pairs (`$w1`, `$w2`) only, not the `phrase` ancestor.

```
($w1,$w2 word) (exists $p phrase):
  ($p ^ $w1) && ($p ^ $w2) && ($w1@orth == $w2@orth) && ($w1 != $w2)
```

Even though `$w1` and `$w2` must be bound to different elements, we still have repetitions of matches where `$w1` and `$w2` are swapped. In this case, the user needs to make his or her choice about the sequential ordering of the two words explicit:

```
($w1,$w2 word) (exists $p phrase):
  ($p ^ $w1) && ($p ^ $w2) && ($w1@orth == $w2@orth) && ($w1 <orth> $w2)
```

The last condition states that `$w1` must precede `$w2` wrt. the orthographic ordering of the `orth` hierarchy. We could also have used the precedence ordering in the offspring tree of the `phrase` ancestor `$p`, which would already imply that `$p` must be a common ancestor of `$w1` and `$w2`:

```
($w1,$w2 word) (exists $p phrase):
  ($w1@orth == $w2@orth) && ($w1 <$p> $w2)
```

Find `words` where subjects *John* and *Mary* speak simultaenously. In this corpus, `word` elements are children of `speaker` elements (the speaker's name is given by the attribute `subj`):

```
($w1,$w2 word) ($s1,$s2 speaker):
  ($s1@subj == "John") && ($s2@subj == "Mary") &&
  ($s1 ^ $w1) && ($s2 ^ w2) && ($w1 # $w2)
```

If we are only interested in words uttered by subject *John* that overlap with someone else's speech, we can apply `exists` quantifiers to all other variables.

```
($w1 word) (exists $w2 word) (exists $s1,$s2 speaker):
  ($s1@subj == "John") && ($s2@subj != "John") &&
  ($s1 ^ $w1) && ($s2 ^ w2) && ($w1 # $w2)
```

TODO: examples for `forall` and subqueries. A simple example is to find a noun phrase where every word overlaps with some noise:

```
($p phrase) (forall $w word) (exists $n noise):
  ($p@cat == "NP") && ($p ^ $w -> $w # $n)
```

Note the complex interplay of universal and existential quantifier, and the importance of the correct ordering (exchanging the declarations of `$w` and `$n` above would give entirely different results – where all words have to overlap with the same `noise` element). The query is much more intuitive with explicit scopes for the quantifiers, and easier for the query processor to optimise

```
($p phrase): ($p@cat == "NP") &&
   {(forall $w word) (exists $n noise): $p ^ $w -> $w # $n}

($p phrase): ($p@cat == "NP") && {
    (forall $w word): $p ^ $w -> { (exists $n noise): $w # $n }
  }
```

A more tricky problem is to find a hesitation (`hesit`) of subject *John* where nobody else is trying to barge in (a rare phenomenon in most group discussions :–). For the solution, we negate a scope with two existential quantifiers:

```
($h hesit) ($s1 speaker):
  ($s1@subj == "John") && ($s1 ^ $h) && ! {
     (exists $s2 speaker) (exists $w word):
     ($s2@subj != $s1@subj) && ($s2 ^ $w) && ($h % $w)
  }
```

Note how this generalises to any speaker when we omit the `($s1@subj == "John")` test.

7.3 Examples of queries embedded in stylesheets

These examples show how NiteQL queries will be embedded in XSLT stylesheets. All code samples below are fragments of stylesheets. In particular, the XML and XSLT declarations are omitted. The output method is assumed to be plain text.

The first stylesheet simply prints all sentences from a corpus as textual transcription, enclosed in `<s>` and `</s>` tags. Note that a NiteQL query used in a `match` attribute must specify a condition for the context element `$$` (and not return any other variable bindings).

```
<xsl:template match="/">
  <!-- the top-level template extracts all <s> elements -->
  <xsl:apply-templates select="($s s)"/>
</xsl:template>

<!-- for each <s> element, output all <word> ancestors -->
<xsl:template match="($$ s)">
  <!-- the query above is equivalent to XPath match="s" -->
```

```
    <xsl:for-each select="($$ s) ($w word): $$ ^ $w">
      <!-- use XPath to output attribute value -->
      <xsl:value-of select="./@orth"/>
    </xsl:for-each>
  </xsl:template>
```

This example will only work properly if the `word` elements (and also the `s` elements) are returned in precedence order. Otherwise, the words have to be sorted in the stylesheet, e.g. by their start times:

```
  ...
    <xsl:for-each select="($$ s) ($w word): $$ ^ $w">
      <xsl:sort select="@nite:start" data-type="number"/>
      <xsl:value-of select="./@orth"/>
    </xsl:for-each>
  ...
```

A second example creates lists of gestures for each gesture type. Only the template which walks through the type hierarchy and compiles the lists of gesture elements is shown.

```
  <xsl:template match="($$ gtype)">
    <entry label="{@name}">
      <list>
        <xsl:apply-templates mode="insert_link"
             select="($$ gtype) ($g gest): $g >'TYPE' $$"/>
      </list>
      <xsl:apply-templates select="($$ gtype) ($t gtype): $$ ^1 $t"/>
    </entry>
  </xsl:template>
```

7.4 Queries based on the NOM corpus example

Find instances where the single subject gesticulates with both hands at the same time.

```
  ($g1,$g2 gest):
    ($g1@hand == "left") && ($g2@hand == "right") && ($g1 # $g2)
```

Because of the definition of the time operators, this query will also return matches where the gestures overlap almost, but not quite (i.e. the second gesture starts *exactly* when the first ends). Add a `+` modifier to enforce a noticeable amount of overlap, or a `++` modifier for substantial overlap.

```
  ($g1,$g2 gest):
    ($g1@hand == "left") && ($g2@hand == "right") && ($g1 #++ $g2)
```

Use pointer relations to find possessive pronouns and their antecedents (indicated by a pointer with role `ANTECEDENT`). We assume here that the antecedent may be either a noun phrase (`np`) or a single word (`word`).

```
($w word) ($ante np | word):
  ($w@pos == "PP$") && ($w >"ANTECEDENT" $ante)
```

Extract verb+object pairs from the syntactic annotations, in the form of a verb and a noun phrase which are children of the same verb phrase. The `vp` parent should not be part of the query result. We use the regular expression `/VB.*/` to identify words tagged as verbs in the Penn Treebank tagset.

```
($w word) ($np np) (exists $vp vp):
   ($w@pos ~ "VB.*") && ($vp ^1 $w) && ($vp ^1 $np)
```

Find occurrences of the word *these* with an H* accent, overlapping with the stroke phase of a deictic gesture. The gesture type is indicated by a pointer with role `TYPE`, pointing to a `gtype` element in the hierarchy of gesture types.

```
($g gest) ($p phase) ($w word) (exists $a accent) (exists $t gtype):
  ($w@orth == "these") && ($a@tobi == "H*") && ($a ^ $w) &&
  ($p # $w) && ($p@type == "stroke") && ($g ^ $p) &&
  ($g >"TYPE" $t) && ($t@name == "deictic")
```

Had we used a more generic gesture type, such as *topographic*, we would not have found gestures marked as *deictic*. The pointer-to-subtree operator can be used to express such generalisations in a type hierarchy in the way below.

```
($g gest) ($p phase) ($w word) (exists $a accent) (exists $t gtype):
  ($w@orth == "these") && ($a@tobi == "H*") && ($a ^ $w) &&
  ($p # $w) && ($p@type == "stroke") && ($g ^ $p) &&
  ($g >"TYPE"^ $t) && ($t@name == "topographic")
```