WQuery 0.4.0

User Guide (outdated)

Marek Kubis

WQuery 0.4.0: User Guide (outdated)

Marek Kubis

Copyright © 2007-2011 Marek Kubis

Table of Contents

List of Tables

List of Examples

Preface

WQuery is a domain-specific query language designed for WordNet-like lexical databases. The WQuery interpreter operates on platforms that provide Java Runtime Environment and works with wordnets stored in XML files. It may be used as a standalone application or as an API to a lexical database in Java-based systems.

This document is a work in progress. Some parts are marked by the *To Be Written* label. The document refers to WQuery 0.5.1. The complete user guide will be distributed with WQuery 1.0.0.

Chapter 1. Getting started

Prerequisites

WQuery requires Java Runtime Environment (JRE) version 1.5 or higher. The latest JRE can be found at http://java.sun.com/javase/downloads/index.jsp .

Download

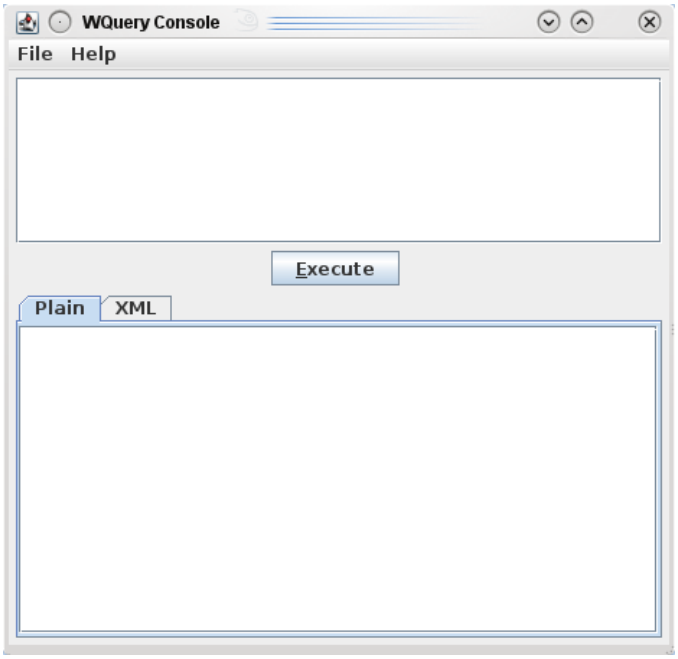
The latest version of WQuery may be downloaded from http://www.wquery.org/download.html.

Installation

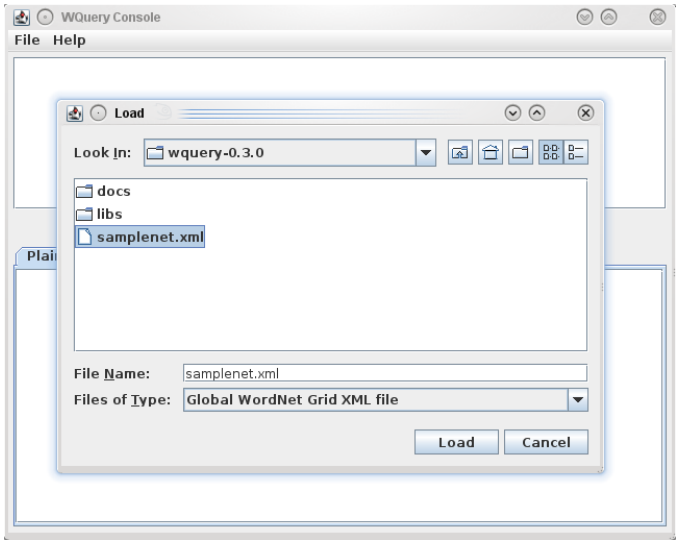
Unzip the downloaded package anywhere. WQuery is ready to use.

Running the console

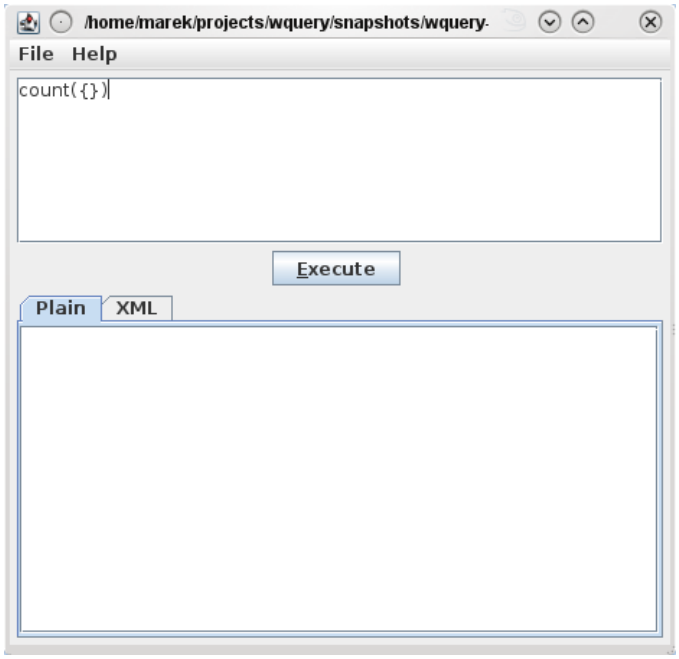
Change into the unzipped directory and run `wguiconsole.bat` (`wguiconsole` on Linux). You will see the following screen.



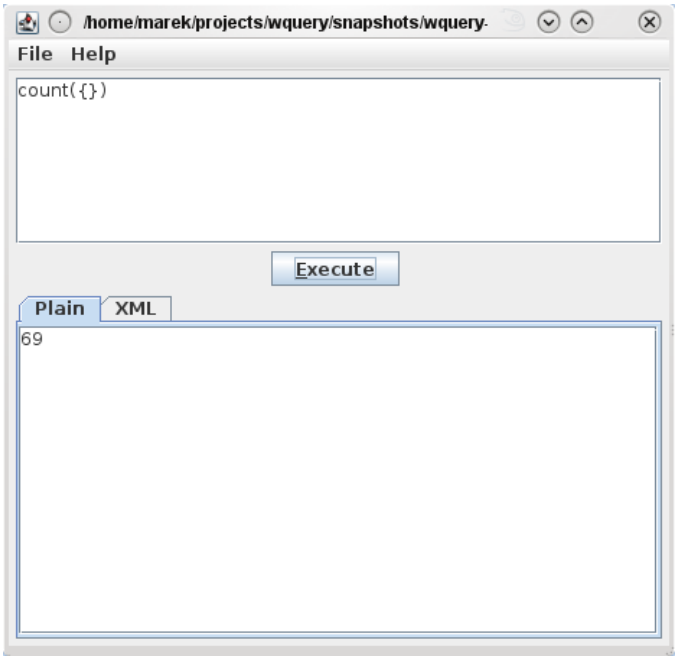
From the *File* menu pick the *Load Wordnet...* option and choose the file that contains a wordnet.



Type `count({})` into the text field in the upper part of the window in order to find how many synsets exist in the loaded wordnet.



Click the *Execute* button or hit **Alt+E**. A result will appear in the text field in the lower part of the window.



WQuery basics

WQuery operates on wordnet related terms like synsets, word senses and words. For example to check whether an ontology[1] contains the word *car* one may simply write

```
car
```

The system answers

```
car
```

If you type the string *a1b1c1d1* which is not a proper word form in SampleNet the system will answer

```
(no result)
```

If you put *a1b1c1d1* in backquotes like in the query below

```
`a1b1c1d1`
```

The system will not treat *a1b1c1d1* as a word form but as an arbitrary character string and will return it to the output.

To check if SampleNet contains the second noun sense of *person* you may type

```
person:2:n
```

In order to find all synsets that contain the word form *person* one may surround that word with curly brackets

```
{person}
```

[1]For the purpose of this section it is assumed that you have loaded an ontology from the file `samplenet.xml` distributed with WQuery. This ontology has been derived from Princeton WordNet by selecting synsets which are used by WQuery test suite (see `samplenet.xml` for Princeton WordNet licensing conditions).

The system returns

```
{ person:1:n individual:1:n someone:1:n somebody:1:n mortal:1:n soul:2:n }
{ person:2:n }
{ person:3:n }
```

As you see synsets are represented in the output as lists of word senses surrounded with curly brackets.

Beside wordnet-specific data types you may also use in queries:

- integers

  ```
  1 -7
  ```

- floating point numbers

  ```
  3.4 5e-2
  ```

- boolean values

  ```
  true false
  ```

Datasets[2] may be transformed using dot operator followed by a relation name.[3] To find all hypernyms of synsets that contain the word form *person* you may type

```
{person}.hypernym
```

and to find all glosses of those hypernyms you may submit the following query

```
{person}.hypernym.desc
```

If you want to retrieve hypernyms of *person* synsets together with their glosses you have to surround chosen path steps with the selection signs < and >.

```
{person}.<hypernym>.<desc>
```

By repeating dots you may apply a relation to the step that is located before the one that precedes the relation step. For example to find *person* synsets together with their part of speech symbols and glosses you may write

```
<{person}>.<pos>..<desc>
```

or

```
<{person}>.<desc>..<pos>
```

Note: Available relations

Relations are extracted from tags found in the wordnet file. For every subtag *T* of every *SYNSET* tag found in the wordnet file the following rules are applied to extract relations:

1. If T equals *ILR* then the value of the *type* attribute becomes a relation name and the content of T is interpreted as a synset identifier to which the relation points.

2. If T equals *LITERAL* or *SYNONYM* or *WORD* it is skipped.

[2] A dataset in WQuery is a bag (multiset) of objects that share the same type.
[3] Such expressions are called paths. Parts of paths separetad by sequences of dots are called steps.

3. Otherwise the lowercased name of T becomes a relation name and the content of T is interpreted as:

   - a synset identifier
   - or as a number (if it is not a valid synset identifier)
   - or as a boolean value (if it is not a valid number)
   - or as a character string (if it is not a valid boolean value).

Additional relations may be infered using relational operators. For example to find all hyponyms of synsets that contain the word form *var* you may precede the *hypernym* relation with the inverse operator ^.

```
{car}.^hypernym
```

To find all transitive hypernyms of synsets that contain the word form *car* you may succeed the *hypernym* relation with the transitive closure operator *!*.

```
{car}.hypernym!
```

To find all holonyms of synsets that contain the word form *car* regardless of the holonymy type you may combine holonymy relations using the union operator |.

```
{car}.partial_holonym|member_holonym
```

More pleasant names may be assigned to relations by using the assignment operator =.

```
hypernyms=hypernym
hyponyms=^hypernym
holonyms=partial_holonym|member_holonym
gloss=def
```

Datasets may be filtered by providing a conditional expression between *[* and *]* signs. To find all noun synsets that contain the word form *car* you shall type

```
{car}[type = `n`]
```

A filter is applied separately to every element of the predeceasing dataset. You may refer to the element which is passed to the filter by using the back reference operator *#*. The query below returns all hypernyms of synsets that contain the word form *person* except the one that contains the second noun sense of the word form *being*.

```
{person}.hypernym[# != {being:2:n}]
```

WQuery provides a set of built-in functions that operate on values of datasets. For example to count synsets that contain the word form *person* you may use the function *count* as shown below.

```
count({person})
```

The complete list of built-in functions may be found in Appendix B, *Built-in functions*.

Paths may be combined together using dataset union, intersection and difference. You may find all *person* synsets hypernyms together with all *car* synsets by using the *union* operator.

```
{person}.hypernym union {car}
```

The Cartesian product of datasets returned by two paths may be constructed using the comma operator.

```
{person}, {car}
```

Beside the expressions presented above WQuery also has a set of imperative expressions like loops, if-statements and assignments. The complete description of all WQuery expressions may be found in Chapter 2, *Language reference*.

Next steps

Read Chapter 2, *Language reference* in order to master the WQuery language. Read Chapter 3, *Advanced topics* to learn how to customize WQuery and embed it into your own applications. Consult Appendix B, *Built-in functions* for a detailed list of functions available in WQuery.

Chapter 2. Language reference

Data types

Basic data types

There are six basic data types in WQuery: booleans, integers, floats, strings, word senses and synsets.

Booleans have one of two logic values either *true* or *false*. They are represented in query results as two literals shown below.

```
true false
```

Integers are represented in query results as strings of decimal digits optionally prefixed with `-` sign.

```
123 -4 576 0
```

Floats (floating point numbers) are represented as strings of decimal digits followed by dot and at least one decimal digit. Floats may also be prefixed with `-` sign.

```
3.0 -1.2 0.0 2.345
```

Strings (character data) are passed to the output without any modifications.

```
apple person man-eating shark
```

Word senses are represented as triples that consist of a word form, a sense number and a part of speech symbol joined together with colons.

```
apple:1:n cold:2:a entail:3:v
```

Synsets are represented as lists of word senses joined together with whitespaces and surrounded with `{` and `}` signs.

```
{ apple:2:n orchard apple tree:1:n Malus pumila:1:n }
```

Tuples

Tuples are finite, ordered collections. Every element of a tuple has to be an instance of a basic data type. It is not possible to create a tuple that contains another tuple as an element.

A tuple is represented in a query result as a single line that consists of representations of tuple elements joined together with spaces. For example a tuple that consists of the synset `{apple:2:n}` followed by `{person:3:n}` and the word form *car* is written as

```
{ apple:1:n } { person:3:n } car
```

Relations

Let X and Y mean sets of values of basic data types T and U. Every subset of Cartesian product of X and Y is called a *relation* with *predecessor type* T and *successor type* U.

WQuery refers to relations stored in a wordnet by their names. Table 2.1, “Obligatory relations.” describes relations that are available in every wordnet loaded into WQuery. Additional relations are usualy retrieved as described in *Note: Available relations* in Chapter 1, *Getting started*.

Table 2.1. Obligatory relations.

Relation	Predecessor type	Successor type	Meaning
a `id` *b*	word sense	string	*b* is an identifier of word sense *a*
a `id` *b*	synset	string	*b* is an identifier of synset *a*
a `sensenum` *b*	word sense	integer	*b* is a sense number of word sense *a*
a `senses` *b*	string	word sense	*a* is a word form of word sense *b*
a `senses` *b*	synset	word sense	*b* is a word sense of synset *a*
a `synsets` *b*	string	synset	*a* is a word form of synset *b*
a `synset` *b*	word sense	synset	*a* is a word sense of synset *b*
a `words` *b*	synset	word sense	*b* is a word form of synset *a*
a `word` *b*	word sense	string	*a* is a word form of word sense *b*

Generators

A generator is an expression that represents a dataset of objects sharing the same basic data type.

Literals `true` and `false` generate datasets that contain exactly one boolean value.

Example 2.1. Boolean generators

```
wquery> true
true
wquery> false
false
```

A string of decimal digits generates a dataset that contains exactly one integer value. Two strings of decimals joined together with `..` operator generate a dataset that contains a range of integers.

Example 2.2. Integer generators

```
wquery> 123
123
wquery> 1..5
1
2
3
4
5
```

A string that represents a floating point number generates a dataset that contains exactly one float.

Example 2.3. Float generators

```
wquery> 12.3
12.3
wquery> 12e3
12000.0
wquery> 12.3e3
12300.0
```

The content of the multiset generated by a character string depends on signs that surround the string.

- A dataset generated by a character string enclosed in single quotes contains that string if and only if it is a valid word form in the loaded wordnet. Otherwise the generated dataset is empty.
- A dataset generated by a character string enclosed in back quotes always contains the surrounded string.
- A dataset generated by a character string enclosed in double quotes contains every word form from the loaded wordnet that matches a regular expression defined by the surrounded string.
- A dataset generated by a character string that is not surrounded with any quotation marks has the same content as a dataset generated from that string enclosed in single quotes.

Example 2.4. String generators

```
wquery> 'person'
person
wquery> `zzz333`
zzz333
wquery> zzz333
(no result)
wquery> "^zymol"
zymology
zymolysis
zymolytic
wquery> person
person
```

Note: Ordering in the output

One may notice in the example above that word forms generated by the `"^zymol"` expression are sorted in the output. Sorting is by default applied to all query results except those that are generated by constructs descibed in the section called “Imperative expressions”. Also if there existed duplicated elements in the query result they would be removed before passing the result to the output.

One may generate all word forms stored in the loaded wordnet by submitting two single quote signs without any characters between then.

Example 2.5. Generating all word forms

```
wquery> ''
(...)
```

In order to generate a dataset that contains at most one word sense one may join a character string, a sense number and a part of speech symbol with colons. The generated multiset will be empty if the loaded wordnet does not contain requested word sense.

Example 2.6. An unique word sense generator

```
wquery> 'person':1:n
person
wquery> 'zzzz':23:n
(no result)
```

If we omit in the expression above the second colon and the part of speech symbol then all word senses with the given word form and sense number will be generated.

Example 2.7. A non-unique word sense generator

```
wquery> set:2
set:2:n
set:2:v
set:2:s
```

All word senses stored in the loaded wordnet may be generated by submitting two colon signs.

Example 2.8. Generating all word senses

```
wquery> ::
(...)
```

By surrounding an expression that returns a dataset of strings with `{` and `}` signs one may obtain a dataset that consists of synsets that contain at least one word form represented by the enclosed expression.

Example 2.9. A word forms to synsets generator

```
wquery> {orange}
{ orange:2:n orangeness:1:n }
{ orange:1:n }
{ orange:3:n orange tree:1:n }
{ orange:4:n }
{ orange:1:s orangish:1:s }
wquery> {"^zymol"}
{ zymology:1:n zymurgy:1:n }
{ zymosis:1:n zymolysis:1:n fermentation:2:n fermenting:1:n ferment:3:n }
{ zymotic:1:a zymolytic:1:a }
```

The same holds for datasets of word senses.

Example 2.10. A word senses to synsets generator

```
wquery>
wquery> {apple:2:n}
{ apple:2:n orchard apple tree:1:n Malus pumila:1:n }
wquery> {apple:2}
{ apple:2:n orchard apple tree:1:n Malus pumila:1:n }
```

All synsets stored in the loaded wordnet may be generated by submitting `{` and `}` signs without any expression between them.

Example 2.11. Generating all synsets

```
wquery> {}
(...)
```

Paths

A path consists of a generator followed by zero or more transitions. Each transition begins with one or more dots followed by a relation name. The generator and the following transitions are called *steps*. The result of applying a transition that consits of *k* dots followed by the relation name `R` to an *n*-th step (*1* <= *k* <= *n*) expression on the left of the transition is a dataset `{b| a in D and R(a,b)}` where `D` is:

1. A dataset returned by the generator of the path if *k* = *n*.

2. A result of applying from the left to the right *n* - *k* transitions to the generator of the path if *k* < *n*.

For example to find all hypernyms of a synset that contains the word form *car* in its first noun sense one may write

Example 2.12. A simple path

```
wquery> {car:1:n}.hypernym
{ motor vehicle:1:n automotive vehicle:1:n }
```

and to find all senses of hypernyms of the synset above one may write

Example 2.13. Another simple path

```
wquery> {car:1:n}.hypernym.senses
motor vehicle:1:n
automotive vehicle:1:n
```

By surrounding chosen path steps with `<` and `>` signs one may retrieve results of intermediate transitions. The result of applying selectors to the chosen *k* steps consists of *k*-element tuples such that the *i*-th element (*1* <= *i* <= *k*) of a tuple belongs to the dataset generated by a subpath defined by all steps of the path that precede the i-th `>` sign on the path. For example to find all triples that consist of a synset that contains the word form *car* followed by its identifier and an identifier its hypernym one may write

Example 2.14. Selectors

```
wquery> <{car}>.<id>..hypernym.<id>
{ cable car:1:n car:5:n } 102934451 103079741
{ car:1:n auto:1:n automobile:1:n machine:6:n motorcar:1:n } 102958343 103791235
{ car:2:n railcar:1:n railway car:1:n railroad car:1:n } 102959942 104576211
{ car:4:n elevator car:1:n } 102960352 103079741
{ car:3:n gondola:3:n } 102960501 103079741
```

As shown in the example above consecutive tuple elements are separated in the output with single spaces.

Relational expressions

A relational expression consists of one or more relations combined together with operators shown in Table 2.2, “Relational operators.”. The relational expression may be used in a transition instead of a single relation.

In order to find all transitive hypernyms of synsets that contain the word form *person* one may write

Example 2.15. A transitive closure

```
wquery> {person}.hypernym!
{ entity:1:n }
{ physical entity:1:n }
{ organism:1:n being:2:n }
{ causal agent:1:n cause:4:n causal agency:1:n }
{ human body:1:n physical body:1:n material body:1:n soma:3:n build:2:n
    figure:2:n physique:2:n anatomy:2:n shape:3:n bod:1:n chassis:1:n
    frame:3:n form:5:n flesh:2:n }
{ grammatical category:1:n syntactic category:1:n }
```

Table 2.2. Relational operators.

Operator	Expression	Result
inversion	`^r`	*(b,a)* such that *(a,b)* belongs to `r`
transitive closure	`r!`	*(a,b)* such that *(a,b)* belongs to `r` or there exist a_1, ..., a_k such that *(a,*a_1*)*, *(*a_1*,*a_2*)*, ... *(*a_k*, b)* belong to `r`
union	`r|q`	*t* such that *t* belongs to `r` or `q`
intersection	`r&q`	*t* such that *t* belongs to `r` and `q`

By using `=` operator one may assign a name to a relational expression and use it in the following queries. For example one may define an alias for transitive hypernymy

```
thyper=hypernym!
```

and use it as shown in the query below

Example 2.16. A transitive closure alias

```
wquery> {person}.thyper
{ entity:1:n }
{ physical entity:1:n }
{ organism:1:n being:2:n }
{ causal agent:1:n cause:4:n causal agency:1:n }
{ human body:1:n physical body:1:n material body:1:n soma:3:n build:2:n
    figure:2:n physique:2:n anatomy:2:n shape:3:n bod:1:n chassis:1:n
    frame:3:n form:5:n flesh:2:n }
{ grammatical category:1:n syntactic category:1:n }
```

Filters

A filter is an expression placed after a path step to select some elements from the generated dataset. The filter consists of a conditional expression surrounded with `[` and `]` signs. Each element of the dataset being filtered is passed separately to the filter and may be referenced inside it by using # sign.

To find all synsets that contain the word form *person* except the one that contains that word form in its thirdh noun sense one may write

Example 2.17. A simple filter

```
wquery> {person}[# != {person:3:n}]
{ person:1:n individual:1:n someone:1:n somebody:1:n mortal:1:n soul:2:n }
{ person:2:n }
```

Objects that precede on the path the element that is analyzed in the current pass may be referenced by repeating # sign.

..... The following query returns all hyponyms of synsets that contain the word form *person* having the same number of words as their hypernyms one may wrtie

Example 2.18. Backward reference

```
wquery> {person}.hypernym[count(#.words)<count(##.words)]
{ organism:1:n being:2:n }
{ causal agent:1:n cause:4:n causal agency:1:n }
```

A reference may be omitted if it consists of exactly one # sign and is followed by at least one step. The expression written above may be rephrased as follows

Example 2.19. Implicit reference

```
wquery> {person}.hyponyms[count(words) = count(##.words)]
{ organism:1:n being:2:n }
{ causal agent:1:n cause:4:n causal agency:1:n }
```

A filter may also be used independently as a boolean generator.

Example 2.20. Filter generator

```
wquery> [1 < 2]
true
```

The following subsections describe three types of operators that are allowed in filters.

Dataset-oriented conditional operators

The operators listed in Table 2.3, “Dataset-oriented conditional operators.” are binary operators that compare pairs of datasets.

Table 2.3. Dataset-oriented conditional operators.

Operator	Expression	Result
dataset equality	`x = y`	true iff datasets generated by `x` and `y` are equal
dataset inequality	`x != y`	true iff datasets generated by `x` and `y` are not equal
dataset inclusion	`x in y`	true iff the dataset generated by `x` is a subset of the one generated by `y`
dataset proper inclusion	`x pin y`	true iff the dataset generated by `x` is a proper subset of the one generated by `y`

Example 2.21, “Dataset comparisons” consists of queries that involve dataset-oriented conditional operators.

Example 2.21. Dataset comparisons

```
wquery> [{person:1:n} = {person}]
false
wquery> [{person:1:n} in {person}]
true
wquery> [{person:1:n} pin {person}]
true
wquery> [{person} != {person}]
false
```

Value-oriented conditional operators

The operators listed in Table 2.4, “Value-oriented conditional operators.” are binary operators that are able to compare only those datasets that contain at most one element. Compared elements have to belong to the same data type.

The following rules hold while comparing two values:

- Numbers are compared using natural ordering.
- Strings are compared by assuming lexicographical order.
- The boolean value `false` is assumed to be lesser than `true`.
- The result of comparing synsets or word senses is undefined.[1]

Table 2.4. Value-oriented conditional operators.

Operator	Expression	Result
lesser than	`x < y`	true iff `x` and `y` contain exactly one element and the element in `x` is lesser than the one in `y`
lesser than or equal	`x <= y`	true iff `x` and `y` contain exactly one element and the element in `x` is lesser than or equal to the one in `y`
greater than	`x > y`	true iff `x` and `y` contain exactly one element and the element in `x` is greater than the one in `y`
greater than or equal	`x >= y`	true iff `x` and `y` contain exactly one element and the element in `x` is greater than or equal to the one in `y`

Example 2.22, “Value comparisons” consists of queries that involve some of value-oriented conditional operators.

Example 2.22. Value comparisons

```
wquery> [person < individual]
false
wquery> [1 <= 1]
true
wquery> [true > false]
true
```

Logic operators

The operators listed in Table 2.5, “Value-oriented conditional operators.” may be used to combine conditional expressions.

Table 2.5. Value-oriented conditional operators.

Operator	Expression	Result
conjunction	`x and y`	true iff both `x` and `y` are true
disjunction	`x or y`	true iff `x` or `y` is true
negation	`not x`	true iff `x` is not true

Example 2.23, “Logic operations” presents expressions that involve logic operators.

Example 2.23. Logic operations

```
wquery> {}[car in words and gondola in words]
{ car:3:n gondola:3:n }
wquery> {}[car in words or gondola in words]
{ cable car:1:n car:5:n }
{ car:1:n auto:1:n automobile:1:n machine:6:n motorcar:1:n }
{ car:2:n railcar:1:n railway car:1:n railroad car:1:n }
{ car:4:n elevator car:1:n }
{ car:3:n gondola:3:n }
wquery> {}[car in words and not gondola in words]
{ cable car:1:n car:5:n }
{ auto:1:n automobile:1:n }
{ car:2:n railcar:1:n railway car:1:n railroad car:1:n }
{ car:4:n elevator car:1:n }
```

Functions

A call to a function consists of a function name followed by an expression surrounded with parentheses. WQuery functions fall into two categories:

aggregate functions — These functions are invoked directly on a dataset returned by the expression between parentheses.

scalar functions — These functions are invoked separately for every tuple that belongs to a dataset returned by the expression between parentheses and the result is a sum of results returned by all invocations.

The list of all built-in functions may be found in Appendix B, *Built-in functions*.

Example 2.24. Sample scalar functions

```
wquery> upper({car:1}.words)
AUTO
AUTOMOBILE
CAR
MACHINE
MOTORCAR
wquery> length({car:1}.words)
3
4
7
8
10
```

Example 2.25. Sample aggregate functions

```
wquery> count({car:1}.words)
5
wquery> max({car:1}.words)
motorcar
```

Arithmetics

Arithmetic expressions in WQuery utilize paths and functions that return integers and floats.

Example 2.26. Sample arithmetic operations

```
wquery> 2 + 3
5
wquery> count({car:1}.words) + count({car:3}.words)
7
wquery> max(length({car}.words)) - min(length({car}.words))
9
```

Operators available in arithmetic expressions are shown in Table 2.6, “Arithmetic operators.”.

Table 2.6. Arithmetic operators.

Operator	Expression	Result
unary minus	`-x`	negation of `x`
unary plus	`+x`	unchanged value of `x`
addition	`x + y`	sum of `x` and `y`
subtraction	`x - y`	difference of `x` and `y`
multiplication	`x * y`	product of `x` and `y`
division	`x / y`	quotient of `x` and `y`
modulo	`x % y`	remainder from division of `x` by `y`

Path expressions

A path expression consists of one or more paths combined together using operators shown in Table 2.7, “Path operators”.

Table 2.7. Path operators

Operator	Expression	Result
path union	`x union y`	union of datasets generated by `x` and `y`
path intersection	`x intersect y`	intersection of datasets generated by `x` and `y`
path difference	`x except y`	difference of datasets generated by `x` and `y`
path product	`x, y`	a dataset that consists of all tuples (a_1, ..., a_k, b_1, ..., b_n) such that (a_1, ..., a_k) belongs to a dataset generated by `x` and (b_1, ..., b_n) belongs to a dataset generated by `y`

Example 2.27. Path expressions

```
wquery> {car:1:n} union {car:3:n}
{ car:1:n auto:1:n automobile:1:n machine:6:n motorcar:1:n }
{ car:3:n gondola:3:n }
wquery> {car} intersect {car:1:n}
{ car:1:n auto:1:n automobile:1:n machine:6:n motorcar:1:n }
wquery> {car} except {car:2:n}
{ cable car:1:n car:5:n }
{ car:1:n auto:1:n automobile:1:n machine:6:n motorcar:1:n }
{ car:4:n elevator car:1:n }
{ car:3:n gondola:3:n }
wquery> {apple}, {set}
{ apple:1:n } { set:2:n }
{ apple:1:n } { determine:3:v set:2:v }
{ apple:1:n } { fixed:2:s set:2:s rigid:5:s }
{ apple:2:n orchard apple tree:1:n Malus pumila:1:n } { set:2:n }
{ apple:2:n orchard apple tree:1:n Malus pumila:1:n } { determine:3:v
    set:2:v }
{ apple:2:n orchard apple tree:1:n Malus pumila:1:n } { fixed:2:s
    set:2:s rigid:5:s }
```

Imperative expressions

Path expressions described in the previous section may be used to construct five types of imperative expressions: *emissions*, *iterators*, *conditionals*, *blocks* and *assignments*.

Emissions

```
emit path_expr
```

An emission passes tuples generated by the path expression `path_expr` to the output.

Example 2.28. An emission

```
wquery> emit {person:1:n}.words
soul
individual
somebody
someone
mortal
person
```

Iterators

```
from var_decls in path_expr imp_expr
```

An iterator executes the imperative expression `imp_expr` for every tuple generated by the path expression `path_expr`. Variables from the comma separated list `var_decls` are substituted with consecutive elements of a tuple processed in the current pass. Those variables may be used as generators in `imp_expr`.[2]

[2]Variable names are prefixed with $ sign.

Example 2.29. An iterator

```
wquery> from $a in {person}.words emit $a
person
soul
individual
somebody
someone
mortal
person
person
```

In the example above the imperative expression built from an iterator and an emission returns a dataset that is not sorted and contains duplicated elements. In order to sort a dataset one have to use the `sort` function.

Example 2.30. The `sort` function

```
wquery> from $a in sort({person}.words) emit $a
individual
mortal
person
person
person
somebody
someone
soul
```

Duplicates may be removed from a dataset by the `distinct` function.

Example 2.31. The `distinct` function

```
wquery> from $a in distinct(sort({person}.words)) emit $a
individual
mortal
person
somebody
someone
soul
```

Conditionals

```
if path_expr imp_expr
```

A conditional executes the imperative expression `imp_expr` if the path expression `path_expr` is *true*.

Note: Truth in WQuery

A path expression is considered to be false if it generates a dataset that is either empty or includes exactly one tuple that consists entirely of boolean elements and at least one of those elements is *false*. Otherwise the expression is considered to be *true*.

Example 2.32. A conditional

```
wquery> if [2 + 2 = 4] emit `ufff...`
ufff...
```

A conditional may have also an optional else block which is executed if `path_expr` is not *true*. The resulting conditional looks as follows

```
if path_expr imp_expr else else_expr
```

Example 2.33. An else block

```
wquery>
    if [bus in {car}.words]
        emit {bus}
    else
        emit `no "bus" word found in "{car}.words"`

no "bus" word found in "{car}.words"
```

Blocks

```
do imp_expr_1 ... impr_expr_n end
```

A block executes sequentially imperative expressions `imp_expr_1` to `imp_expr_n`.

Example 2.34. A block

```
wquery>
do
    emit {person:3:n}
    emit car:2:n
    emit apple
end

{ person:3:n }
car:2:n
apple
```

Assignments

```
var_decls = path_expr
```

An assignment binds variables from the comma separated list `var_decls` to consecutive values of a tuple generated by the path expression `path_expr`. The `path_expr` expression shall return exactly one tuple.

Example 2.35. An assignment

```
wquery>
do
    $a, $b = <{car:1:n}>.<desc>
    if [$b =~ `engine`]
             emit $a.words
end

machine
motorcar
automobile
auto
car
```

Variable bindings are accessible in all expressions that follow the assignment and are placed in the scope of the block which surrounds it. Bindings in the inner block hide the ones defined in the outer blocks.

Example 2.36. Hidden binding

```
wquery>
do
    $a, $b = 1, 2

    do
        $b = 3
        emit $a
        emit $b
    end

    emit $a
    emit $b
end

1
3
1
2
```

Chapter 3. Advanced topics

This chapter covers topics important only in specific WQuery use cases.

Embedding the interpreter

WQuery may be easily embedded into your own application. The following libraries must be added to the application classpath if you want to use WQuery:

- `wquery-VERSION.jar`
- `scala-library-2.7.5.jar`
- `slf4j-api-1.5.8.jar`
- `slf4j-log4j12-1.5.8.jar` (or another logging framework facade)
- `log4j-1.2.14.jar` (or another logging framework)

In order to access WQuery programmatically you have to instantiate an object of the class `org.wquery.WQuery` by invoking its static method `getInstance`. After obtaining `WQuery` instance you may execute queries by calling the `execute` method.
Example: To be written

Registering custom wordnet loaders

To be written...

Registering custom functions

To be written...

Using custom emitters

To be written...

Appendix A. Tools reference

WGUIConsole

To be written...

WConsole

To be written...

Appendix B. Built-in functions

Table B.1. Scalar functions.

Function	Argument	Result
abs(x)	*x* - float or integer	absolute value of *x* - float or integer
ceil(x)	*x* - float	the smallest integer that is greater than or equal to *x* - float
floor(x)	*x* - float	the largest integer that is lower than or equal to *x* - float
length(x)	*x* - string	length of *x* - integer
log(x)	*x* - float	the natural logarithm of *x* - float
lower(x)	*x* - string	lowercased *x* - string
power(x,y)	*x* - float or integer, *y* - float or integer	*x* raised to the power of *y* - float or integer
replace(x, y, z)	*x* - string, *y* - string, *z* - string	replace substring of *x* matching regular expression *y* with *z* - string
substring(x, y, z)	*x* - string, *y* - integer, *z* - integer	substring of *x* from *y* to *z* - string
upper(x)	*x* - string	upercased *x* - string

Table B.2. Aggregate functions.

Function	Argument	Result
avg(x)	*x* - string or float	average value of *x* - float
count(x)	*x* - any	*x* count - integer
distinct(x)	*x* - any	set built from values of *x*
max(x)	*x* - string, float or integer	maximal value of *x* - string, float or integer
min(x)	*x* - string, float or integer	minimal value of *x* - string, float or integer
size(x)	*x* - any	*x* tuples sizes
sort(x)	*x* - any	sorted *x*
sum(x)	*x* - float or integer	sum of *x* - float or integer