The inter4ql Interpreter

Łukasz Białek

October 2013

Contents

Introduction **5**

1 Making inter4QL work **7**
1.1 4QL package 7
1.1.1 4QL package structure 7
1.1.2 Ready binary files 7
1.1.3 Documentation 8
1.1.4 Example program files 8
1.1.5 Source code 8
1.1.6 Tests 8
1.2 Compiling the interpreter 8
1.2.1 Linux compilation 8
1.2.2 Windows compilation 9
1.2.3 Windows compilation - known issues 9

2 Using of the interpreter **11**
2.1 Supported commands 11
2.2 Basic usage examples 11

3 Writing own 4QL program **15**
3.1 Module structure 15
3.1.1 Domains 15
3.1.2 Relations 16
3.1.3 Rules 16
3.1.4 Facts 18

4 Developing own inter4QL features **19**
4.1 Class diagram 19
4.2 Data structures 20
4.3 Writing own external module 20
4.4 Class characterizations 21

Summary **25**

A 4QL language grammar **27**

Bibliography **29**

Introduction

4QL is a rule-based query language. Its name is an acronym for *four-valued*[1] *query language*. Development of the language was started in 2010 on Faculty of Mathematics, Informatics and Mechanics (University of Warsaw) and University of Linköping (Sweden) by J. Małuszyński and A. Szałas [2] and is still continued. Two years ago the first version of inter4QL (4QL interpreter) has been developed as part of student master's thesis [3]. Since then, it was used for testing small 4QL programs. It has also been released for public use on project web page[2]. During those two years some problems and bugs has been found, which was the main reason for writing a new version. In September 2013 inter4QL v2.0 was released as part of my master's thesis [1]. Because of constant development of the program, next versions (v2.1 and v2.2) were released afterwards.

It is assumed, that the reader knows basics about interpreter and 4QL language. To find out more about 4QL language, please refer to [2] and [5]. The interpreter description can be found in documents [1] (in Polish, most up-to-date) and [6] (in English). Moreover, there is a full Doxygen [10] documentation included in interpreter source files.

This document is a kind of manual for the newest version of the interpreter. It is also a summary of features added during inter4QL v2.x development. It should give a good background before both using the interpreter and starting own implementation of new features.

[1] This part corresponds to four logical values existing in the language: *true*, *false*, *inconsistent* and *unknown*
[2] 4QL web page: www.4ql.org

Chapter 1

Making inter4QL work

1.1 4QL package

Full inter4QL source code and all additional files can be downloaded from official 4QL web page: www.4ql.org. All news and papers about 4QL language as well as contacts to people working on it can also be found there.

1.1.1 4QL package structure

Let's assume, that the newest 4QL zip package is downloaded from 4QL web page. It can be very easily unzipped using tools available for both Windows and Linux. Let's see the directory structure of the package:

- `bin` - Compiled binary files of the interpreter ready to use
  - `Windows` - Windows binary files
  - `Linux` - Linux binary files
- `doc` - Whole technical documentation of the project
  - `!doxygen_config_file` - Contains configuration file for Doxygen
  - `html` - Contains html documentation of the project
- `examples` - Five examples of valid 4QL program
- `src` - Source code of the project
- `tests` - Tests for the interpreter
  - `algorithm` - 4QL manual test files, testing algorithms
  - `checker` - 4QL manual test files testing checker module
  - `unit` - Automatic tests that can be run automatically by calling `make`.

1.1.2 Ready binary files

The package delivers ready to use binary files with latest version of the interpreter. There are versions compiled for both Windows and Linux. If you do not want to compile the project yourself, you should use those binary files. Except from main program, there are also two

example external module binaries included[1]. However, it can not be guaranteed that those binary files will work on every computer. In case of failure while running the interpreter, you should try to compile the project yourself.

1.1.3 Documentation

The project has full technical documentation available. All class dependencies, function and class descriptions can be found there. Documentation can be viewed by opening `/doc/html/index.html` in any internet browser.

1.1.4 Example program files

In order to better understand 4QL program structure, there are some program files included in package. Those programs can be loaded into the interpreter in order to manually test some features.

1.1.5 Source code

Source code is the main part of inter4QL package. The whole inter4QL is divided into classes. Their structure will be described in chapter 4. In the source code there is a *Makefile* file, which allows to compile the whole project. This file DOES NOT compile example external modules, which source code is in `/src/modules/` directory. There is a separate *Makefile* file to compile those modules in that directory. Whole compilation process will be described in next section.

1.1.6 Tests

The interpreter package is delivered with test files, which allow to check, if the interpreter is fully operational. Tests in `/tests/algorithm/` and `/tests/checker/` directories are manual tests, which means that they have to be manually loaded into the interpreter. First tests should load without any problems. Importing files from checker tests should end with error message. There are also tests in `/tests/unit/` directory, which are automatic tests based on *Makefile* file. To run those tests, you have to call `make` command in the directory of the tests you want to run.

1.2 Compiling the interpreter

The interpreter is ready to be compiled in both Windows and Linux operating systems. *Bison*[7] and *Flex*[9] programs are used for lexical and grammar analysis. Therefore, you have to have *Bison* and *Flex* installed for compilation to be successful.

1.2.1 Linux compilation

First of all, you have to make sure, that *Bison* and *Flex* are installed properly. You will also need `make` program to be installed on your computer. Having all of that done, you should call `make` command in `/src/` directory and wait until compiling process is completed and `inter4ql` binary file is created. All errors during compilation in testing step of inter4QL

[1] More about modules in chapter 3

development were caused by no or improperly installed *Bison* and *Flex*. If you are experiencing any difficulties, please make sure those programs are installed and try reinstalling or updating them.

1.2.2 Windows compilation

Windows compilation has to be done using tools, that emulate Linux environment in Windows. Windows binary files included in 4QL package was compiled using MinGW [11]. Having it installed, you can run MinGW shell and then compile the project just like it should be done in Linux operating system. It is important to install *Bison* and *Flex* during installation of MinGW (those packages can be optional, you have to manually select them in package list).

1.2.3 Windows compilation - known issues

Tests have shown, that there can be some errors during compilation under MinGW even though everything is set up correctly. Those errors are connected with file `lex.yy.cc`, which is generated by *Flex*. Observations have shown, that the problem is because of wrong include section for C++ and lack of "std::" before functions like "cout". After modifying this file by including `iostream` and `cstdio` and adding all missing "std::" (before "cin", "cout", "cerr", "istream" and "ostream") compilation should finish with no problems. Compilation of external modules did not cause any unexpected errors.

Chapter 2

Using of the interpreter

This chapter will present the standard scenario of interpreter usage. It will allow to familiarize yourself with the way it works. It will also present report of *Valgrind*[8], which is responsible for finding memory leaks.

Command used to test the interpreter using *Valgrind*:
`valgrind --leak-check=full --leak-resolution=high --track-origins=yes ./inter4ql`.

2.1 Supported commands

Below you can find commands that are supported in current version of the interpreter:

- `import "FILE_NAME.4ql".` - loads a program from file given as a parameter
- `modules.` - lists all modules available in the interpreter
- `print MODULE_NAME.` - prints out a module, which name is given as a parameter
- `MODULE_NAME.RELATION_NAME(PARAMETERS).` - prints out logical value of requested fact
- `clear.` - resets the interpreter to the initial state
- `exit.` - exits the interpreter.

2.2 Basic usage examples

In file `example.4ql` there is a program, which you can find below:

Listing 2.1: Example progam

```
module example:
  relations:
    r(literal). w(literal). o(literal).
  rules:
    w(x) :- o(x) | r(x).
    r(x) :- w(x).
    -o(x) :- r(x).
  facts:
    o(x).
end.
```

The first step is to import module from example file into the interpreter:

```
        Interpreter for 4ql, version 2.2, http://www.4ql.org/
# import "example.4ql".
Program loaded!
# print example.
module example:
relations:
        r(literal)
        w(literal)
        o(literal)
beginning facts:
        example.o(x (literal)) : true
rules:
        example.w(x (literal)) : true :- example.o(x (literal))
           : true
        |       example.r(x (literal)) : true
        example.r(x (literal)) : true :- example.w(x (literal))
           : true
        example.o(x (literal)) : false :- example.r(x (literal))
            : true
#
```

As you can see parsing and importing the program was successful. There was also `print` command presented. We can now execute some queries for facts - not only to module `example`:

```
# modules.
List of available modules:
------------------------------
- plugin
- math
- example
# example.o(x).
Results:
--------
        example.o(x) : inconsistent
# example.w(VARIABLE).
Results:
--------
        example.w(x) : inconsistent
# math.ltR(5.65, 2.0).
Results:
--------
        math.ltR(5.65, 2) : false
#
```

As you can see, you can use variables in queries. Last query was performed to module built into the interpreter. You can also make queries to external modules. What is important, when you query external or internal module, you can not use variables in the query.

At the end, we use `clear` command and exit the interpreter:

```
# clear.
Modules cleared!
# modules.
List of available modules:
------------------------------
- plugin
- math
# exit.
Thanks for using!
==2478==
==2478== HEAP SUMMARY:
==2478==     in use at exit: 0 bytes in 0 blocks
==2478==   total heap usage: 3,029 allocs, 3,029 frees, 360,684
    bytes allocated
==2478==
==2478== All heap blocks were freed -- no leaks are possible
==2478==
==2478== For counts of detected and suppressed errors, rerun
    with: -v
==2478== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
    from 0)
```

After resetting the interpreter to initial state, there was `modules` command executed one more time to show, that all modules loaded from files are cleared and all modules built into the interpreter are not modified. At the end, you can also see the output of *Valgrind* program ensuring that no memory leaks are possible.

Chapter 3

Writing own 4QL program

Program in 4QL language is a list of *modules*. Modules can be divided into three groups: *local modules*, *internal modules* and *external modules*. All modules declared in 4QL program file are called *local modules*. Internal module is a module built into the interpreter - currently the only example is *math* module. External module is an external application written in a specific way[1].

3.1 Module structure

Module structure has not changed since first version of the interpreter and is described in previously mentioned master's thesis[3]. We can now discuss structure of a module:

Listing 3.1: Schema of module declaration in 4QL program file

```
module NAME:
  domains:
    domain declarations for module NAME
  relations:
    relation declarations for module NAME
  rules:
    rule declarations for module NAME
  facts:
    fact declarations for module NAME
end.
```

3.1.1 Domains

A module can contain zero or more *domains*, which are aliases for internal interpreter types:

- `literal` - alphanumeric type starting with lowercase symbol
- `integer` - signed numeric type
- `string` - string type in quotes

[1]More information about modules can be found in chapter 4

- `real` - signed floating point type
- `logic` - logic type
- `date` - data type in format RRRR-MM-DD
- `datetime` - data and time type in format RRRR-MM-DD GG:MM

Domains are really useful when a relation has some arguments with the same type. They ensure much better readability of the program. What is more, since version 2.1 of the interpreter, domains with same type are not unified, which improves time and memory complexity. Values of domains are also not shared between different modules.

Domain declaration contains type of new domain and its name (example below).

Listing 3.2: Example of domain declarations

```
module example:
  domains:
    literal name.
    date date_of_birth.
    integer height.
  ...
end.
```

3.1.2 Relations

Each module has to contain at least one *relation*. Every relation has to contain at least one parameter. All relations and types of their parameters have to be defined at the beginning of declaration of a module. Module section called `relations` contains declarations of relations in format `relationName(typesOfParameters)`. List of parameters can contain previously defined domains. An example of relation declaration is shown below:

Listing 3.3: Example of relation declaration

```
module example:
  domains:
    string ancestorName.
    string descendantName.
  relations:
    descendant(descendantName, ancestorName).
    age(string, integer).
    younger(string, string).
  ...
end.
```

3.1.3 Rules

Each module can contain one or more rules defined in section `rules`. Each rule consists of one or many *conditions* and just one *conclusion*. Conclusion and conditions are separated by ':-' sequence. Conditions can be separated by comma sign which is equal to logic $\wedge$ sign or

by | sign, which is equal to logic $\vee$. As it was mentioned before, there are four logic values in 4QL language - *true, false, inconsistent* and *unknown*. Because of that, logic operators have to be redefined:

$\wedge$	f	u	i	t
f	f	f	f	f
u	f	u	u	u
i	f	u	i	i
t	f	u	i	t

$\vee$	f	u	i	t
f	f	u	i	t
u	u	u	i	t
i	i	i	i	t
t	t	t	t	t

$\rightarrow$	f	u	i	t
f	t	t	t	t
u	t	t	t	t
i	f	f	t	f
t	f	f	t	t

	$\neg$
f	t
u	u
i	i
t	f

Section `rules` contains declarations of rules in format '`conclusion :- conditions.`'.

The most important feature distinguishing 4QL language and languages from `DATALOG`$^{\neg}$[2] *family is possibility to put negation not only in conditions, but also in conclusion of rules.*

Listing 3.4: Example of rule declarations

```
module example:
  ...
  rules:
    younger(NAME, "John") :- age(NAME, X), age("John", Y), math.
        lt(X, Y) | descendant(NAME, "John").
    -olderThanJohn(NAME) :- younger(NAME, "John").
    olderOrPeerOfJohn(NAME) :- -younger(NAME, "John").
  ...
end.
```

As arguments of relation you can use values with types equal to those defined in `relations` section. You can also use *variables*, which values will be deducted automatically from knowledge base. The name of a variable is a string starting with capital letter. Moreover, you can use facts that were defined earlier, in different modules or are defined inside the interpreter (like all relations from *Plugin* and *Math* modules). To use those facts you have to use following structure: `moduleName.relationName(arguments)`.

[2] See [4]

3.1.4 Facts

Last section you have to define while writing module declaration is `facts`, which can contain zero or more *initial facts* for a module. Facts should be defined using structure `relationName(arguments)` and can initially have one of two logical values: *true* or *false*. You cannot use variables in initial facts.

Listing 3.5: Example of initial facts declaration

```
module example:
  ...
  facts:
    age("Evan", 22).
    age("Adam", 30).
    descendant("Chris", "Adam").
    -descendant"Evan", "Adam").
end.
```

Chapter 4

Developing own inter4QL features

This chapter will introduce internal structure of the interpreter which hopefully will make start of developing new features as easy as possible. We will start with class diagram, then all classes will be described and, at the end, some ideas about future development will be presented.

4.1 Class diagram

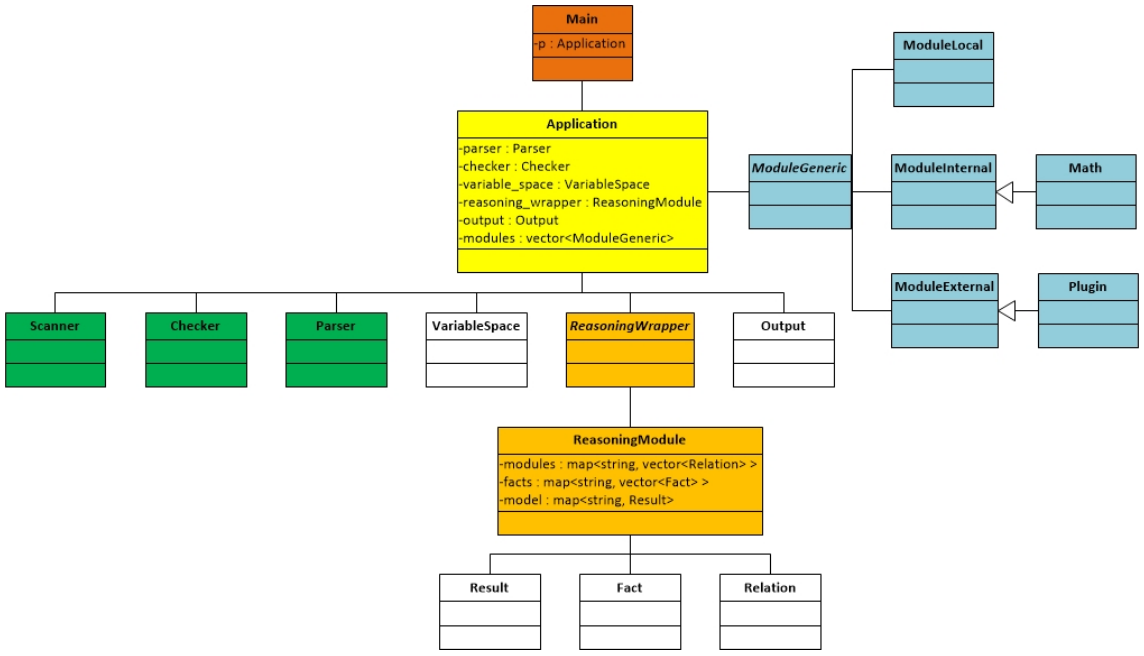

Figure 4.1: (Simplified) class diagram of inter4QL

This diagram is presented here to give a brief view on how the interpreter is built. The block colored with red color is the main class of the project. It contains only one field of type `Application`. `Application` class is colored with yellow color. It is where main loop of the program is executed. All I/O handling, computation executing and main memory management is done in that class. It also manages all modules, colored with blue color on the diagram. There is one, general class for all types of modules called `ModuleGeneric`. Next, there are three classes for each type of module: `ModuleLocal`, `ModuleExternal` (which is

inherited by `Plugin`) and `ModuleInternal` (which is inherited by `Math`). Green boxes show all classes, that are responsible for lexical and grammar analysis of 4QL programs (`Scanner` and `Lexer`) as well as `Checker` class, which is responsible for other program checks, that are not possible during first step of program parsing (just like type checks). At the end, there are also orange classes, which are responsible for all calculations inside the interpreter (those classes contain whole knowledge base and algorithms used to calculate *well supported model*[1]).

4.2 Data structures

Changing algorithms may create a need to change data structures used to store data in the interpreter. That is why the most necessary data structures are described below:

- `map<string, vector<Relation> > modules` - it is a map, which key contains name o a module and value contains a vector o relations existing in that module (each relation contains its name and types of parameters)
- `map<string, vector<Fact> > modules` - the key of this map contains name of a module and value contains a vector o initial facts for this module (each fact contains information about name of its module and relation as well as logic value of the fact)
- `map<string, Result > model` - the key of this module contains again name of a module. Value stores Result object. This object keeps all facts that exist in well supported model for a module. It also has implemented methods of adding new facts to the module model.

All structures described above are filled with data at the end of parsing a program in `Application` class. They are used by algorithms that generate well supported model.

4.3 Writing own external module

Every external application can be used as an external module. It has to meet following conditions:

1. It has to be placed in `./plugins/ directory`
2. It has to react in specific way to commands presented below.

All commands, which external module application should implement:

- `interface` - the application should print out types of parameters, which should be passed to it. Valid types are: `LITERAL, INTEGER, STRING, REAL, DATE, DATE_TIME, LOGIC`
- `calculate` - this command should be followed by parameters (number and types of parameters were specified by `interface` command). Then it should perform calculations and return a result in form of exit code (1 - true, 0 - false)
- `help` - it is an optional command, it helps to manage external modules

[1] All about well supported model can be found in fallowing document: [5]

In the inter4QL package there are two sample external modules included: *lt* and *contains*. First one implements simple function *less than* with interface `INTEGER INTEGER`. Another one has interface `STRING STRING` and it returns *true* if first parameter is a substring of second one. The easiest way to develop own plugin is to copy source code of any existing one and modify it the way you want it to work.
You can see below, how *lt* module reacts to commands presented above:

```
lukasz@lukasz-PC /usr/src/src/plugins: ./lt help
Inter4ql plugin - less than

Usage: Put into Plugins in Inter4ql directory and run
   interpreter. This plugin will be automatically recognized and
    installed.

lukasz@lukasz-PC /usr/src/src/plugins: ./lt interface
INTEGER INTEGER

lukasz@lukasz-PC /usr/src/src/plugins: ./lt calculate 2 3

lukasz@lukasz-PC /usr/src/src/plugins: echo \$?
1
```

4.4 Class characterizations

There is no point in rewriting documents [1], [5] and [6] and describe one more time all algorithms implemented in the interpreter. The code is also self-explanatory enough to understand all procedures. This section will present every class in the interpreter with general information what it is responsible for (obviously, it is not possible to describe each class in very accurate way). This information should allow to find out what classes have to be modified (more or less) just by knowing a feature that has to implemented.

- `Application` - the main loop of the program

  Managing input lines

  Scanning program for values and adding them to domains

  Clearing memory (class destructor)

- `Checker` - checking program for problems that can not be checked earlier

  Finding cycles in queries in modules

  Checking duplicated names of modules, relations, domains

  Checking facts and rules for correctness of parameters (number and type)

  Checking if all referenced modules exist

- `Disposable` - list of objects to be freed when parsing error occurs

  Managing only predefined types of objects

  Adding object to object list

Clearing list in error mode (with freeing objects)

Clearing list in normal mode (just freeing a list in destructor)

- `Domain` - stores domain data type

  Keeping domain name

  Keeping domain type

- `Exception` - custom exception class

  Keeping exception string

  Keeping `Disposable` object (to be ale to clear objects during exception handling)

- `Fact` - stores data structures for facts

  Keeping fact module name

  Keeping fact relation name

  Keeping vector of `Value` objects representing parameters

  Keeping fact logic value

  Keeping info about set values (in facts like `MODULE.RELATION(PARAMS) in {true, false, inconsistent, unknown}`)

  Clearing memory in normal mode (deleting set values)

- `Functions` - global functions used in different parts of the interpreter

  Converting parsing error location to error message

  Converting string to variable type object

  Converting variable type object to string

- `LineParser.y` - input file for *Bison* program

  Adding created objects to `Disposable` object

  Managing 4QL grammar

- `Logic` - functions implementing logic operators

- `Main` - main class of the interpreter

  Creating `Application` object and running main loop

- `Math` - class containing all operations for this internal module

  Adding (programmatically) relations to the module based on function that are available

  Executing functions based on relation name

- `ModuleExternal` - general class for external modules

  Execute call to external application from `Plugin` class

  Clear `Relation` objects in destructor

- `ModuleGeneric` - the most generic class for modules

  Every change in this class has to be reflected in all types of modules

- `ModuleInternal` - general class for internal modules
  - Execute function calls from `Math` class
  - Clear `Relation` objects in destructor
- `ModuleLocal` - general class for local modules
  - Adding facts to reasoning module
  - Managing default domains in local module (inbuilt interpreter types)
  - Perform queries for facts to reasoning module
  - Clear data used in local modules in destructor
- `Output` - wrapper for output operations
  - Keeping stream responsible for std/err output operations
- `Parser` - parsing wrapper class
  - Managing communication with `Scanner` and `LineParser` classes
- `Plugin` - class containing all operations for external module
  - Scanning for external application that could be used as a module
  - Running module application and gathering results
- `Program` - class used to store successfully parsed 4QL program
  - Initiating all objects that are part of program (Relations, Domains...)
- `ReasoningModule` - module performing all reasoning operations
  - Generating facts with logic value *unknown* only for referenced facts
  - Adding and removing modules in knowledge base
  - Adding facts to knowledge base
  - Finding facts matching query fact
  - Generating well supported model (algorithm based on document [5])
  - Clearing data used in reasoning process
- `ReasoningWrapper` - wrapper allowing to change reasoning modules
- `Relation` - stores data structures for relations
  - Keeps name of relation and module name
  - Keeps vector of `Domain` objects describing relation parameter types
- `Result` - object used to store well supported model
  - Keeps vector of `Fact` objects
  - Implements fact adding method to chenge fact logic value if needed
  - Implements equality method for reasoning algorithm
  - Implements clearing method for destroying fact in model

- `Rule` - stores data structures for rules

  Keeps one fact for rule conclusion and vector of vector of fact for conditions

  Clears data used in rule

- `Scanner.l` - input file for *Flex* program

  Contains all definitions for lexems (like numbers, strings etc)

- `Types.h` - contains definitions of all major types

  Defining type of variables (main interpreter types)

  Defining logic values

  Defining command types and parsing data structure

- `Value` - stores data structures for values

  Keeps information about the value of the object depending on its type

- `VariableSpace` - object containing information about domains

  Keeps domains separately for each module

  Clearing values from domains in destructor

Those are all classes and files that are included into newest version of the interpreter. Based on those characterizations, you can easily track classes that has to be changed while implementing new features. For example when you plan to implement new way of calling external modules, you know that you have to reimplement module application and modify `Plugin` class. You may need to make some additional small changes in `ModuleExtarnal` , which you can also deduct from data written above.

Summary

This document is a brief manual for the latest version of inter4QL. It was written for all people, who have not ever worked with 4QL language or interpreter to familiarize themselves with the idea and working implementation. In the introduction also was presented some articles about 4QL and inter4QL, which you can read to find out more. Those papers with this one create really good background for logic professionals and software developers who want to work in and develop 4QL world.

Appendix A

4QL language grammar

This appendix contains full grammar of 4QL language in *Extended Backus-Naur Form*. All non terminal symbols are placed in triangular brackets while terminal ones are placed in quotes and are written with `fixed-width font`. ϵ symbol means empty production.

$$
\begin{array}{rcl}
< program > & ::= & < modules > \\
< modules > & ::= & \epsilon \mid < modules > \mid < module > \\
< module > & ::= & \texttt{'module'} < id > \texttt{':'} \\
 & & < domains >< relations >< rules >< facts > \texttt{'end.'} \\
< domains > & ::= & \epsilon \mid \texttt{'domains:'} \quad < domain\_declarations > \\
< domain\_declarations > & ::= & \epsilon \mid < domain\_declarations >< domain\_declaration > \\
< domain\_declaration > & ::= & < type > \texttt{'} \; \texttt{'} < id > \texttt{'.'} \\
< relations > & ::= & \epsilon \mid \texttt{'relations:'} \quad < relation\_declarations > \\
< relation\_declarations > & ::= & \epsilon \mid < relation\_declarations >< relation\_declaration > \\
< relation\_declaration > & ::= & < id > \texttt{'('} < parameters > \texttt{').'} \\
< rules > & ::= & \epsilon \mid \texttt{'rules:'} \quad < rule\_declarations > \\
< rule\_declarations > & ::= & \epsilon \mid < rule\_declarations >< rule\_declaration > \\
< rule\_declaration > & ::= & < rule\_head > \texttt{':-'} < relation\_list > \texttt{'.'} \\
< rule\_head > & ::= & < id > \texttt{'('} < args > \texttt{').'} \mid \texttt{'-'} < id > \texttt{'('} < args > \texttt{').'} \\
< relation\_list > & ::= & < relation\_in\_rule > \\
 & & \mid < relation\_list > \texttt{','} < relation\_in\_rule > \\
 & & \mid < relation\_list > \texttt{'|'} < relation\_in\_rule > \\
< relation\_in\_rule > & ::= & < id > \texttt{'('} < args > \texttt{').'} \\
 & & \mid \texttt{'-'} < id > \texttt{'('} < args > \texttt{').'} \\
 & & \mid < id > \texttt{'.'} < id > \texttt{'('} < args > \texttt{').'} \\
 & & \mid \texttt{'-'} < id > \texttt{'.'} < id > \texttt{'('} < args > \texttt{').'} \\
< args > & ::= & < arg > \mid < args > \texttt{','} < arg > \\
< arg > & ::= & < value > \mid < variable > \\
< facts > & ::= & \epsilon \mid \texttt{'facts:'} \quad < fact\_declarations > \\
< fact\_declarations > & ::= & \epsilon \mid < fact\_declarations >< fact\_declaration > \\
< fact\_declaration > & ::= & < id > \texttt{'('} < values > \texttt{').'} \\
 & & \mid \texttt{'-'} < id > \texttt{'('} < values > \texttt{').'} \\
< values > & ::= & < value > \mid < values > \texttt{','} < value > \\
< value > & ::= & < id > \mid < string > \\
 & & \mid < integer > \mid < real > \\
 & & \mid < logic > \mid < date > \mid < datetime > \\
< type > & ::= & \texttt{'literal'} \mid \texttt{'string'} \mid \texttt{'integer'} \\
 & & \mid \texttt{'real'} \mid \texttt{'logic'} \mid \texttt{'date'} \mid \texttt{'datetime'} \\
< parameters > & ::= & < id > \mid < parameters > \texttt{','} < id >
\end{array}
$$

Productions mean:

- $< id >$ - string of literals starting with lower-case sign
- $< string >$ - string of literals in quotes
- $< integer >$ - signed number
- $< real >$ - signed floating point number (with dot as a separation mark)
- $< logic >$ - value from set $true, false, unknown, incons$
- $< date >$ - data in format RRRR-MM-DD
- $< datetime >$ - data with time in format RRRR-MM-DD GG:MM

Bibliography

[1] L. Bialek. Rozwoj interpretera regulowego jezyka zapytan 4QL, M.Sc. Thesis, MIM UW, 2013.

[2] Jan Maluszynski i Andrzej Szalas, Living with Inconsistency and Taming Nonmonotonicity. Datalog Reloaded, LNCS 6702, 384-398, Springer-Verlag, 2011.

[3] Patryk Spanily: Interpreter czterowartosciowego regulowego jezyka bazodanowego 4QL, M.Sc. Thesis, Wydział Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, 2011.

[4] Serge Abiteboul, Richard Hull, Victor Vianu: Foundations of Databases. Addison-Wesley 1995.

[5] Jan Maluszynski i Andrzej Szalas, Partiality and Inconsistency in Agents' Belief Bases. KES-AMSTA, Frontiers of Artificial Intelligence and Applications 252, 3-17, IOS Press, 2013.

[6] Patryk Spanily: The inter4QL interpreter, 2012 (http://4ql.org/wp-content/uploads/2012/10/inter4ql.pdf)

[7] *Bison - GNU parser generator*, http://www.gnu.org/software/bison/

[8] *Valgrind*, http://valgrind.org/

[9] *flex: The Fast Lexical Analyzer*, http://flex.sourceforge.net/

[10] *Doxygen*, http://www.stack.nl/~dimitri/doxygen/

[11] *MinGW*, http://www.mingw.org/