

NyQuiL: A Numeric Query Language

Konstantin Voevodski and Benjamin N. Waber
Computer Science Department, Boston University,
111 Cummington Street Boston, MA 02215, USA
{kvodski, bwabes@cs.bu.edu}

1 Introduction

This technical report describes the development of the Numeric Query Language, or NQL (also known as NyQuiL). While this project began with the rather modest goal of merely extending the query parser created by Stephen Metsker, it later grew into the creation of a useful and to our knowledge unique programming language.

While the GUI (Graphical User Interface) was slightly improved to allow us to save queries and results, open queries and later databases, this was by no means the focus of our project. After all, designing a GUI is simply a matter of programming simple widgets, while designing a language takes considerably more time and skill.

From the humble beginnings of a language that could only interpret one query at a time in a simple format, NQL has blossomed into a programming language rife with variable definitions, recursion, the common flow of control constructs, the support for multiple queries, and much more. The possible extensions, many trivial, foretell a bright future for NQL. Internal and external function definitions come to mind, but also user-defined arrays, support for stream handling, and the capability to modify database entries are all open for future groups to explore.

The true power a language like NQL offers is astounding. Imagine, if you will, NQL programs replacing the common framework of databases interfaces on the web. Instead of the clumsily link-

ing web pages and individual SQL queries using Javascript, the future NQL could do all this in a single program using only NQL. No longer would programmers have to be conversant in both of these languages to effectively write so-called Active Server Pages. It could all be done using the language called NQL.

1.1 Overview

The first phase of the project was deciding in what direction to take the query language originally developed by Metsker. Our first idea was to extend Metsker's parser to recognize and evaluate numeric queries such as sort, average, and standard deviation. We found Metsker's query language to be deficient in several other ways. Some of the problems we saw was the ability to only issue one query at a time, as well as the fact that the database that the original parser came with was hard-coded into the package. The language that Metsker's parser recognized also did not support recursion. In the following weeks, we addressed these problems and extended the original package with other useful features.

Our first improvement was an extension of the original GUI, which allows the user to load and save queries, and save query results. We then worked on allowing our language to recognize recursive queries. Once this was accomplished, the next logical step was to allow the user to evaluate multiple queries at once by inserting semicolons

in between successive queries.

Still, we felt that NQL was not robust enough to be considered a viable query language. Since the user now had the ability to evaluate multiple queries, it seemed that the user would benefit from being able to define variables that he/she could use repeatedly. We also noticed that there was a distinction between using strings in quotation marks and strings without quotation marks within queries. This is due to the fact that Metsker's parser considers quoted strings to be field values as opposed to class or variable names. We therefore gave the user two types of variables to choose from when programming, a var type corresponding to unquoted strings, and a string type corresponding to quoted strings. Up until this point we had always been querying on the hard-coded Chips database provided by Metsker. Our language would never be useful if the user was not able to query on his/her own databases, without having to significantly modify the source code. To that end, we developed an intuitive syntax which allows the user to define a custom database. We created objects to make newly defined databases available to the query engine, first checking that the database satisfied our syntax specifications.

It also came to our attention that there was an oversight in the previous NQL versions which allowed the user to define variables whose names matched reserved keywords such as select, from, and where, as well as class and field names of the current database. We corrected this problem by checking to make sure that each variable which was used had a name distinct from any of the reserved words.

We still had not yet realized our initial goal of supporting common numeric operations such as sorting and average, among others. The sort command was implemented first, followed by min, max, and median. Next came sum, average, and standard deviation. This allowed the user to perform statistical analysis operations on portions of a database.

Noticing that our language still lacked the basic control flow structures common to most programming languages, we implemented support for if/else statements as well as for-loops. It was now possible to write more complex programs in the style of an imperative programming language.

The final portion of our project consisted of documenting our code and reorganizing our software package into objects which grouped functions in a more intuitive way. We organized our utility functions according to which objects they were used by, leaving only the most general functions in the utility class. In addition, we created Javadocs for each object which we created or worked with. This will allow other programmers to more easily understand our code in the future.

1.2 Schedule

Week 1 - GUI Enhancements

Week 2 - Recursion and multiple evaluation

Week 3 - Side-effects

Week 4 - Mutable Databases

Week 5 - Numeric operations

Week 6 - Support for reserved keywords

Week 7 - Control flow structures

Week 8/9 - Documentation and Testing

1.3 Tasks

Tasks among NQL team members were divided as follows: Konstantin Voevodski was responsible for recursion, mutable databases, and numeric operations, while Benjamin Waber was responsible for side-effects, flow of control, the GUI, and the project website. Both team members contributed to the NQL documentation.

1.4 Technical Report Organization

This technical report is organized in the following way: section 2 contains the formal presentation of the syntax that NQL recognizes and in-

interprets, section 3 describes the functionalities that NQL supports, section 4 provides justifications for the extensions that we made, section 5 gives a description of our package organization, section 6 describes how to activate and use our parser/interpreter and gives several NQL programs, section 7 presents our testing methods and results, and section 8 contains our final thoughts on the project.

2 Syntax

Figure 1 contains the numeric extensions to the formal grammar recognized by Metsker's parser, where `select` is defined as it is by Metsker, `|selectTerm|` indicates that the corresponding `select` contains that `selectTerm`, and `{select}` indicates that the query is on strictly one `selectTerm`.

Similarly, figure 2 shows the programming extensions to the formal grammar recognized by Metsker.

All queries must be terminated by the ";" character. The syntax of `arithmeticExpression` is the same as the syntax presented by Metsker for "expression" in the arithmetic parser. `classNames` represents all the class names of the current database. `classFields` represents all the field names of the current database.

3 Functionalities

NQL has the functionalities developed by Metsker at its core, which allow the user to query based on a class name, some (one or more) field names, and some (zero or more) restrictions (reflected in the where clause). To clear some confusion, Metsker's query parser does allow for multiple statements in the where clause, despite what has been stated by one of the query groups during their presentation. The parser treats all strings as field or class names, unless they are in quotations, which indicates that the string is a field value. All numbers are treated as field values.

The language has a relatively simple user interface consisting of a query window, a results window, and a metadata window, as well as two buttons: "go" for evaluating a query, and "clear" for clearing the results window. A drop-down menu is also available allowing the user to open and save queries, save results of a query, and load a custom database. A help menu is available as well, which gives the address of the website devoted to NQL.

Our language allows the user to write programs as opposed to individual queries. Each statement within the program is terminated by the ";" character. NQL will only look to evaluate queries terminated in such manner, and will output error messages if the queries are not properly terminated. NQL also recognizes if statements and for loops. Each if, else, and for statement must be terminated by the ";" character, which comes right before the body of each conditional. Because the syntax of the conditionals must match the one recognized by the Logikus parser, all logic expressions must be in prefix notation.

NQL also allows imperative operations, supporting two primitive types: `val` and `string`. The difference between the two is that a `string` type is treated as a field value (as opposed to a class/field name) by the parser. Note that the declarations

```
string temp = Safflower; val temp = "Safflower";
```

are equivalent. Each variable must be given a type upon initialization which does not need to be re-stated when the value of the variable is changed, unless the user intends to change its type. The environment class which handles variable and type assignments has a `repOk` function which makes sure that after each imperative operation every variable has some value and a valid type. Error messages are outputted if the user attempts to initialize a variable without listing its type, or if the type is invalid. NQL prohibits the user from declaring variables which

```

numExpression = sort | min | max | median | stddev | sum | average
sort = "sort (" select ")" by" |selectTerm|
min = "min (" select ")" by" |selectTerm|
max = "max (" select ")" by" |selectTerm|
median = "median (" select ")" by" |selectTerm|
stddev = "stddev (" {select} ")"
sum = "sum (" {select} ")"
average = "average (" {select} ")"

```

Figure 1: Numeric syntax extensions. Here, `select` is defined as it is by Metsker, `|selectTerm|` indicates that the corresponding `select` contains that `selectTerm`, and `{select}` indicates that the query is on strictly one `selectTerm`.

have the same name as any of the language keywords or any of the class/variable names of the database currently in use.

Another feature which NQL supports is recursion, which is recognized by the parser through the presence of parenthesis within queries. The evaluation strategy for recursion is to first parse and evaluate the text within the innermost set of parenthesis, return the result, and repeat (assuming the previous query did not yield an error) until all parenthesis are eliminated. If any query within the recursive sequence is invalid, an error message listing that query is outputted to the query result window.

NQL also allows the user to load and query on custom databases. A "Load Database" option is available from the drop-down menu which allows the user to specify a file containing the database that he/she would like to query on. Any new database must be defined according to syntax which NQL recognizes: metadata declaration must begin with `<begin metadata>` and end with `<end>`, while data declaration must start with `<begin data>` and end with `<finish>`. Also, no blank lines are allowed within the metadata and data declarations. Each variable and fact must be

separated by a comma and each functor is separated from its data by a set of parenthesis. If any of these prerequisites are not met by a new database, NQL outputs an appropriate error message either in the metadata window or in the query result area (depending on when it recognizes that the syntax of the database file is incorrect).

NQL is also capable of recognizing and interpreting numeric queries such as `sort`, `min`, `max`, `median`, `average`, `standard deviation`, and `sum`. The user may issue a query and then add one of the numeric query commands on top of it, parenthesizing the query and putting the appropriate command keyword immediately in front of it. Of course, in order for these commands to make sense, the `average`, `standard deviation`, and `sum` commands may only be used with queries on a single numeric field, while the `sort`, `min`, `max`, and `median` commands must specify a field by which the comparisons are to be performed (because it is possible to issue those commands on queries on multiple fields). The only other requirement for the latter commands is that the specified field must be a queried field that is also numeric.

```

comparison = arg operator "(" select ")" | "(" select ")" operator arg |
arg operator arg
variable = Word | varName
arg = expression | varName
varName = Word \ (classNames U reservedWords U classFields)
reservedWords = "select" | "from" | "where" | "min" | "max" | "sort"
| "average" | "median" | "stddev" | "sum" | "for" | "if" | "val" |
"string"
optionalType = empty | "var" | "string"
equalsExpression = optionalType varName "=" select |
numExpression | arithmeticExpression
optionalElse = empty | "else; {" body "}"
ifComparison = operator "(" arg "," arg ")"
body = (select | equalsExpression | numExpression | ifExpression |
forExpression)*
ifExpression = "if" ifComparison ";" "{" body "}" optionalElse
forExpression = "for (" equalsExpression | select | numExpression ","
ifComparison ","
equalsExpression | select | numExpression ")"
singleton = varName

```

Figure 2: Programming syntax extensions.

4 Extensions

NQL extends the language originally recognized by Metsker by allowing the user to write query programs as opposed to individual queries, program with side-effects, query on different databases, have greater querying power, and combine querying and analysis of data.

NQL allows the user to have several statements parsed/evaluated at the same time by inserting a delimiting character (;) in-between the statements. The user also has greater programming power because of the ability to use if statements and for loops. Side-effects are introduced in order to let the user write programs which are easier to think about. NQL supports a string type and a val type, a distinction which proves useful in not having to put quotations around string values in queries. NQL guarantees that every variable has a valid name, some value, and a valid type. A valid name is a name different from all reserved words such as NQL commands (select, sort, etc.) as well as variable and class names of the database that was used when the variable was initialized. Each variable's initialization and type annotation occur at the same time. Once a variable has a type, it need not be restated in subsequent assignments, unless the programmer would like to change its type, which is allowed.

We also allow the user to query on different databases, which gives great flexibility to our parser. Metsker's parser came with a hard-coded and primitive database, together with many objects for just generating the variable and fact structures for the query engine to use. Clearly, the language recognized by his parser could never be fully functional because the user would need to restructure a considerable portion of the code (significantly modify several files) in order to query on a different database. NQL lets the user load his/her own database, by selecting a "Load Database" GUI command, and then selecting the desired database file ("Load Database" is one of the items that we added to the GUI, which

now has a pull-down menu). The syntax of the database file that NQL recognizes is fairly intuitive and familiar to anyone who has programmed with functors. Once a "Load Database" command is issued, the database object first makes sure that the database file contains proper syntax. If so, the database metadata is displayed in the metadata window of the GUI and a "New Database Loaded" comment appears in the query result area of the GUI. If the syntax of the database is not proper, an error message displaying what is wrong is displayed in the result area. The session always begins with the default database being the "data.txt" file, which is in the same folder as the run command. If that file does not contain a valid database, or is missing entirely, an error message is displayed in the metadata window when the GUI initially pops up on the screen. After this, the user is free to load any new database. Also note that each loaded database may contain several classes. For example, Metsker's chip database can be loaded from one file, even though it contains chip, order, and customer "tables." In fact, for the sake of familiarity, the default database file does contain the original chip database. Once the user queries on the new database, the build function of QueryBuilder uses the database object's function to again make sure the database that is currently "loaded" contains proper syntax. The build function then uses the database object to obtain variable and fact structures to build the query from what the user has inputted.

NQL also allows the user to define recursive queries, something which was missing from Metsker's original parser. Recursion in a query allows the user to obtain information which would otherwise be unavailable using Mesker's parser. Using the original chip database as an example, suppose that the user wanted to know the first and last name of a person who ordered a particular chip. Unfortunately, the order table has neither the chip name, nor the first nor the last name of the customer who makes the order. The order table has only the chip id, and the customer

id. So one way to obtain the answer using the original parser would be to first query the chip id based on the chip name from the chip data, write it down, then use that chip id to get the customer id from the order data, write that down again, and then finally query the customer first and last name from the customer data based on customer id. Notice that in our new version the user could have used side-effects to store these intermediary results. Yet it would be foolish not to let the user obtain this information in one step. The problem can be solved with one recursive query such as

```
select LastName, FirstName from Customer
where CustomerID = (select CustomerID from
Order where ChipId = (select ChipID from Chip
where ChipName = "FillInName"))
```

This extension allows the user to have significantly greater power in terms of querying ability.

To go along with the ability to define recursive queries, we also needed more precise error messaging to let the user know why a particular query cannot be evaluated. Consider the above query, for example. It could easily result in an error, yet contain perfect syntax. If more than one customer orders the same chip, this query is nonsensical because after two evaluations of the innermost parentheses the engine is asked to return a last name and a first name from the customer table where customerId is equal to two different values. NQL, if faced with this situation, will output an error message as well as the part of the recursive query that goes wrong, as opposed to the whole query, because we do not expect the user to be particularly insightful in the queries that are submitted.

Finally, NQL enhances Metsker's parser by letting the user combine querying and statistical analysis. These extensions were the first functionalities which we envisioned our improved language supporting. Since we named the language before we did a lot of the work on it, its name

is still an acronym for Numeric Query Language even though it now supports much more than just querying and numerical analysis. We believe that numeric analysis extensions are useful because statistical tools are always needed to analyze database fields, especially when the databases are very large. This point is emphasized by the use of students' grades from CS511 as one of the databases to showcase and test NQL, because this database naturally lends itself to statistical analysis which NQL now allows the user to perform instantly (such as querying on who got the highest grade, the lowest grade, what the standard deviation, median, and mean were on any of the quizzes / midterm).

NQL supports min, max, median, sort, sum, average, and stddev commands which the user may add on top of queries in order to analyze a particular portion of the database. Note that sum, average, and stddev commands are only valid for queries on one field, which must also be numeric. Min, max, median, and sort, however, may be performed on queries on several fields, as long as the user specifies according to which field the comparisons are to be made. Thus NQL requires every min, max, median, and sort command to specify a certain field. For example,

```
min (select id, quiz5 from student) by quiz5
```

will return the id and quiz grade of the person with the smallest quiz5 grade, not the smallest id. Another requirement is that the specified field must be a queried field. This is necessary because NQL uses the query result to do the comparison, and it cannot compare values of a field which is not present. Of course, error messages are outputted if the user violates any of the rules that NQL upholds for numeric queries.

Interfaces

Name	File	Description
Speller	Speller.java	This interface defines the role of a speller that returns the proper spelling of a class or variable name, given any spelling.

Classes

Name	File	Description
ClassNameAssembler	ClassNameAssembler.java	Pops a class name and informs a QueryBuilder that this is a class to select from.
ComparisonAssembler	ComparisonAssembler.java	This assembler pops a comparison term, an operator, and another comparison term.
ComparisonParser	ComparisonParser.java	This utility class provides support to the NQL parser, specifically for <code>expression()</code> and <code>comparison()</code> subparsers.
environment	environment.java	The environment class is used to store variable names, types, and values.
forConstruct	forConstruct.java	Interprets for statements.
ifConstruct	ifConstruct.java	Interprets if-else statements by returning to JaqlMediator a string containing the correct branch to evaluate as well as the rest of the original query area.
JaqlMediator	JaqlMediator.java	The JaqlMediator class serves as a mediator between the user and other parts of the software package, while also doing a lot of the work associated with recognizing and evaluating sentences of the query language itself.
JaqlParser	JaqlParser.java	This class provides a parser that recognizes a select query, without any where clauses.
JaqlTester	JaqlTester.java	This class tests that NQL can parse random language elements.
JaqlUe	JaqlUe.java	This is a simple user environment (UE) for queries written in the NQL language.

Classes Continued

logikusInterface	logikusInterface.java	This class provides the interface between Logikus and NQL.
MellowSpeller	MellowSpeller.java	This class provides a speller that allows any spelling, which facilitates random testing of NQL language elements.
QueryBuilder	QueryBuilder.java	This class accepts terms, class names and comparisons and then builds a query from them.
SelectTermAssembler	SelectTermAssembler.java	This assembler pops a term and passes it to a query builder.
sortHelper	sortHelper.java	The sortHelper class contains functions used by sortEvaluate in JaglMediator.
structureCreator	structureCreator.java	The structureCreator class is used to create fact and variable structures used by unification engine in evaluating queries.
structureHelper	structureHelper.java	The structureHelper class contains functions used by structureCreator in checking syntax of database files as well as obtaining fact and variable structures.
utility	utility.java	Provides miscellaneous functions for Mediator, Environment, sortHelper, and other classes.
VariableAssembler	VariableAssembler.java	This assembler pops a token from the stack, extracts its string, and pushes a Variable of that name.

Exceptions

Name	File	Description
CannotParseException	CannotParseException.java	Signals that the NQL parser does not recognize expression
MalformedSyntaxException	MalformedSyntaxException.java	Signals that syntax of the database file is malformed
UnbalancedParenthesesException	UnbalancedParenthesesException.java	Signals that expression has unbalanced parenthesis
UnrecognizedClassException	UnrecognizedClassException.java	Signals that a given string is not the name of a known class.
UnrecognizedVariableException	UnrecognizedVariableException.java	Signals that a given string is not the name of a known variable.

Figure 3: File organization and description of class contained in that file.

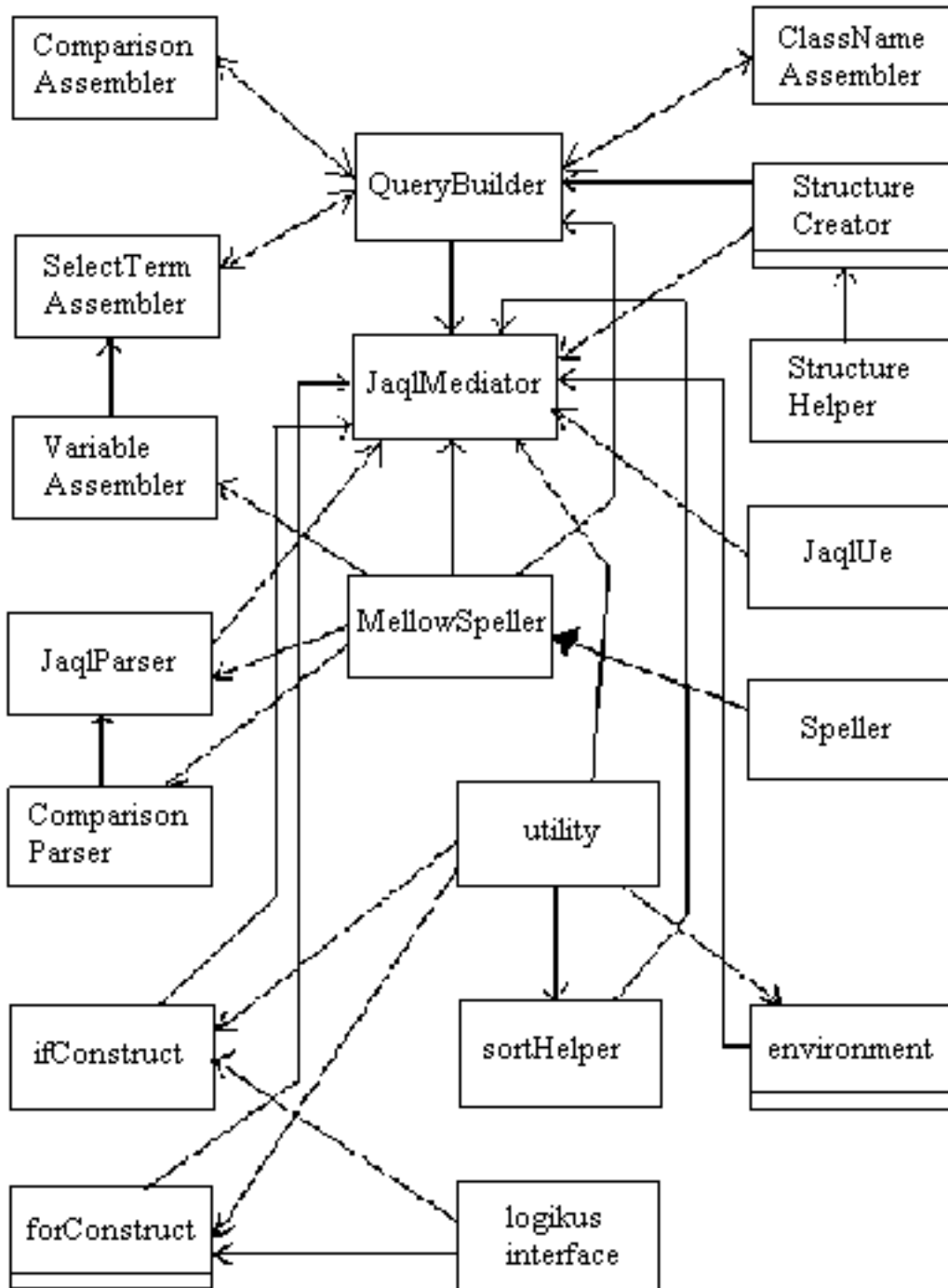


Figure 4: Module dependency diagram of NQL system. A box on the end of a solid arrow denotes a class that implements the interface at the starting end of the arrow. The other arrow type (from module A to module B) means that B uses A. A horizontal bar on the bottom of a box denotes that a class is mutable. All relations in this graph are one to one.

5 Package Organization

The whole of the NQL parser code is located in the folder `sjm/examples/query`. The parser, however, does rely on other code from Metsker's original package, but we did not modify this code in any way. Figure 3 contains the precise content of all Java files in `sjm/examples/query`, while Figure 4 presents the software organization in terms of object dependencies.

6 Running the Package

To run the NQL UE, go to the 'classes' directory of 'NQL' and type:

```
java sjm.examples.query.JaqlUe
```

Queries are issued to the parser through the 'Query' pane of the UE, and query results are displayed in the 'Results' pane. The metadata of the currently loaded database is displayed in the 'Metadata' pane. To execute all queries in the 'Query' pane, press the 'Execute' button. To clear all results, click the 'Clear' button.

Example files may be loaded into the UE by clicking on the 'File' menu and selecting 'Open.' A dialogue box will then appear, allowing the user to browse for an appropriate query file. To save a query or result, select the appropriate 'Save' command from the UE. New databases may be loaded by using the 'Load Database' command. Valid database syntax is as follows:

```
<begin metadata>
database1(field1, field2,...,fieldN)
database2(field1, field2,...,fieldM)
...
<end>
```

```
<begin data>
databasei(data1, data2,...,dataN)
databasej(data1, data2,...,dataM)
```

```
...
<finish>
```

Example databases may be viewed in the 'travelDB.txt', 'gradesDB.txt', or 'testDB.txt' files in the 'Examples' folder within 'NQL'.

Suppose that we have the grades database loaded into NQL. Then, run the NQL program in figure 5. This program looks for the lowest student ID for a student who got greater than or equal to a 70 on the CS 511 midterm. The answer returned is:

```
2462.0;
76.0;
```

Equivalently, we could have run the following query program:

```
val minId = min (sort (select id from student where midterm >= 70) by id );
minId;
select midterm from student where id = minId;
```

And, indeed, we obtain the same answer. Readers are urged to try examples for themselves, as then you will become more proficient in NQL programming and its semantics.

7 Testing

For every extension that we made, we verified that the new version of NQL worked correctly on all test examples that were recognized by the previous version. The underlying theme of our testing methodology was testing with intuitively simpler examples first, gradually moving on to cases where we believed the parser was most likely to fail.

Recursion was tested using queries which contained one, two, and three recursive statements which appeared in different parts of the query.

```

val maxId = max (select id from student) by id;
val minId = min (select id from student) by id;
string temp = dummy;
string noMatches = no data meets query;
val x = 0;
val y = 0;

for (x = minId, <=(x, maxId), x = x + 1);
{
    y = select midterm from student where id = x;
    temp = y;

    if != (temp, noMatches);
    {
        if >=(y, 70);
        {
            x;
            y;
            x = maxId + 1;
        }
    }
}

```

Figure 5: NQL example program on gradesDB.txt database. This program finds the student with the lowest id who got a midterm score that was greater than or equal to 70.

Naturally, queries which contained no recursion were tested as well. Because the implementation of recursion depended largely on string manipulation, several versions of the same query were tested with more/less spaces between the recursive part and neighboring words. The improper syntax handling which came with recursion was also tested, such as checking for unbalanced parenthesis.

Repeated evaluation was tested using queries which contained no statements (no semicolons present), one statement, two statements, and more than two statements. Exception handling, such as determining whether a program contains query-terminating characters, was also tested by providing NQL with programs which either did not contain query-terminating characters (semicolons) all together, or did not contain a query-terminating character at the end of the last query.

We tested support for mutable databases by defining our own databases and then querying on them. The first and last data fields of class objects were queried more extensively, because due to string processing those queries were most likely to yield incorrect results (which they did as bugs were discovered). The exception handling associated with user-defined databases was also verified, such as the handling of loading a database file which does not exist, and the handling of a database file which contains improper format either immediately after the load command or immediately before some query is issued.

NQL's ability to support numeric queries was tested by defining and loading a large database of numeric data (student grades for CS511), and then testing all the numeric queries on that database, comparing some of the results to the grade statistics obtained from the CS511 website. The exception handling associated with statistical queries was also extensively verified, such as the requirement that `stddev`, `average`, and `sum` be used on only queried, single, numeric fields, and that `sort`, `min`, `max`, and `median` commands specify a field which is queried, and also numeric.

Side-effects were also tested in a similar, incremental fashion. First, properly typed variables were used in queries, arithmetic operations, and as singletons. Then, we attempted to assign variables reserved keywords as names, which properly invoked our error handling function. The use of untyped variables and undefined variables was also tested in numerous cases, all of which were handled gracefully by the parser.

We tested the if-else construct by first using a single if, an if followed by an else, and finally multiple nesting of if-else constructs. In all of these cases, we began by testing the if-else statements with only variable definitions preceding it. The conditional branches would output unique values in their particular branch was chosen, and thus it was possible to verify that the correct chain of execution had been taken. We followed this by placing statements after the if-else constructs to verify that all NQL code was being executed. The use of variables in the conditionals themselves were naturally also tested.

For the for-loop testing, we first started with a single for loop, outputting the index value inside the loop and then the final index value after the loop terminated. We similarly employed this method when testing nested for loops, which we thoroughly tested with up to three nestings.

8 Reflection

For the last two months we have worked to give NQL functionalities which would distinguish it from other query languages. Our parser/evaluator provides the user with great flexibility by supporting many useful features, such as the ability to use custom databases and evaluate recursive queries. We have also designed NQL to support a grammar which closely resembles that of an imperative programming language. The use of side-effects and flow of control statements gives the language a different and distinct feel. We did not leave anything half-done, as all expressions

that are recognized by the parser are evaluated as well. However, there are certainly further extensions which other programmers may want to implement, such as the ability to add individual facts to the database as opposed to re-defining the database. Clarifying ambiguities regarding variable scope is another issue which may be addressed by others, should anyone choose to work on this project in the future.

We had to cast a few ideas aside while extending NQL with new features either due to the incompatibility of the potential features to what we had implemented already, or simply due to time constraints. For example, NQL never keeps the database that it is currently working with in memory, instead reading from the database file each time it needs variable or fact structures. An alternate implementation, which would have the database stored in memory, would make it much easier to implement adding or redefining individual fact structures. Also, we thought about adding special constructs which would justify NQL having iterative abilities, such as a variable which can hold a query result and can then be iterated through by a for-loop. However, due to time constraints and the practical ambiguity of such a feature, we chose to not go forward with it.

One of the characteristics of NQL that we were very happy with is its ability to recognize and recover from exceptional situations. Each of the statements in a query program is parsed/evaluated independently, so an error in one of the statements will not affect other statements in the program unless they are directly dependent on the part of the code where the error occurs. NQL outputs precise error messages whenever possible, telling the user exactly what has gone wrong. This is evident in the handling of recursion, side-effects, numeric queries, and loading and querying on a new database. NQL comes very close to only recognizing the expressions which it is supposed to recognize (as far as we know it does, but we also know that testing every possible syntax error for each expression is not possible). These

statements may appear unsatisfactory at first, but we are aware that most projects produced in this class are likely to be flawed because the natural tendency is to only design the language to recognize and interpret sentences that one wants the language to recognize, forgetting about the rest of the inputs (for which the behavior of the program becomes unpredictable).

On a final note, by working on this project we certainly gained valuable experience in terms of working with a large software package and programming in an object-oriented way. We had to deal with understanding and using the code of others as well as trying our best to have our code be understandable to others. Since "object-oriented design" has quickly become the buzzword of the computer science industry, it was nice to gain some experience which may be useful in the future.