2007

Integrating a universal query mechanism into java

Aaron Robinson

Integrating a Universal Query Mechanism into Java

Aaron R Robinson

arr9595@cs.rit.edu

September 8, 2007

Rochester Institute of Technology

The Undersigned Computer Science Faculty

Approves the Thesis

Integrating a Universal Query Mechanism into Java

Dr. Axel T. Schreiner, Chair

Dr. James E. Heliotis, Reader

Dr. Carl H. Reynolds, Observer

Approval Date

Abstract

This thesis discusses design, architecture, and application of a universal query language embedded in Java. Utilizing various design patterns and Java's polymorphism, the current result is a preprocessor that will convert an embedded language into compilable Java. The resulting Java utilizes a back–end developed for the queried data structure, capable of querying that structures internal data.

Acknowledgments

It is an honor and privilege to thank the many people that have helped me to complete this work and that have assisted me in attaining this degree.

I thank my parents for raising me to never question my ability to do anything.

I thank my roommates, Neal V. Lafferty, Graham A. Anthony III, and Nicholas R. Currier, all of which have made my life in Rochester a wonderful journey and who all helped in their own way to get me through my education.

I would also like to thank the staff and professors in the RIT CS department. Particularly, Hans–Peter Bischof for taking a risk and admitting someone with a BA in Social Work and no formal computer skills into the RIT CS Masters program, and Axel T. Schreiner, my thesis chair who offered this thesis to me and who has exposed me to the wonderful world of language processing and compiler design.

Finally, I would like to thank my peers in the CS department, who I consider great friends and with whom I will always be connected to a "network".

You just have to do your best. And if you fail, you just have to do better.

Kingsley Robinson

Contents

Chapter 1

Introduction

The increased organization of data along with the rise of consumable information has led to a need for a simpler and more universal way of sifting through data. The organization of data takes on several forms all of which are made easily accessible via the use of a computer. Databases such as mySQL and XML documents are spread across the Internet, and internal objects within applications may also hold data that a programmer wants. Offering data to the programmer is the difficult part, as getting the information from the aforementioned sources requires different modes of acquisition. A programming solution to this problem was recently debuted by Microsoft, in their release of .NET^{TM} version 3.0: a new mechanism named LINQ[8].

LINQ or *Language INtegrated Query*, uses a syntactic style similar to that of SQL, a well–established way of describing what the programmer wants in a database, to represent where and what data should be returned. The new facility has the ability to look inside of MSIL[1] objects, XML documents, and databases that can integrate with the .NET^{TM} environment [2]. The same query language is used across all three data structures, which enables the developer to focus on the development of the application rather than how to retrieve the data from the particular data structure.

[1]Microsoft Intermediate Language

This concept of querying multiple structures for information could be adopted by other languages such as Java, which is incorporated in many web services and back–end business logic that could benefit from a more unified query tool inherent in the language. The work proposed is to determine how the Java programming language could support a similar system. This required an examination of the back–end logic that makes LINQ run, such as the use of lambda calculus [8], and the development of a preprocessor. The preprocessor takes a defined query language and converts that into compilable Java code, via a back–end class. There is currently a designed back–end[12], which will be used with classes that implement the `Iterable` interface, however, the preprocessor is designed to be generic and will be able to convert the embedded query language into the appropriate back–end that will query the underlying data structure.

Chapter 2

Language Processing

Communication is at the forefront of any relationship and the one between programmer and computer is no less important than that between two human beings. There needs to be a defined way to communicate and also an agreed upon language to be used. Even when this language is defined, it can be a very cumbersome language that both parties are not capable of using easily. Even to the computer a language such as C is too abstract and to many humans machine code is terribly cumbersome to use intelligently. The solution to this problem is an intermediate step that allows a programmer to use a higher level language and the computer to use its own defined language.

The processing of the higher level language into machine code is the responsibility of a compiler and compiler designer. The compiler designer must have a description of the language in which the programmer would like to speak to the computer. This is called a grammar, which is defined as a tuple with the following 4 components [1, page 42]:

1. A set of terminals or tokens
2. A set of non–terminals
3. A set of rules consisting of terminals and non–terminals
4. A start symbol, one of the non–terminals

A grammar can be computed and evaluated in a similar way to that of a mathematical equation. There are different types of grammars that can be constructed but in the realm of compiler design, the designer is most interested in *Context–Free* grammars (Type–2 in the Chomsky hierarchy [6, page 227]). This type is defined simply as having one non–terminal on the left side of a rule and a sequence of terminals and non–terminals on the right [6, page 79]. The grammar defines how strings can be put together in order to create sentences in the language. Meaning is given to the sentences by a separate set of rules later. For now it is enough to say that when the grammar is followed, a sentence will be recognized as being or not being in the language. A closer look at a few parts of the processing are warranted, such as the lexical analysis §2.1, parsing §2.2 and analysis §2.3 of a language that is defined by a grammar.

2.1 Lexical Analysis

Lexical analysis is handled by the scanner, which recognizes the symbols as grammar terminals, *e.g.* "123" as *Number*, and passes this information along with an associated value, in some cases, to the parser. The importance here is to determine what each string in the sentence is. As an example a simple grammar is defined in Figure 2.1 adopted from the Aho Ullman Compiler book [1, page 193]. The scanner will process each character in a string, such as 2.1.

$$2 * (3 \ + \ 4) \tag{2.1}$$

The scanner will return to the parser those identified parts of the string that are defined by the grammar [1, page 109].

$$\begin{aligned} \mathit{sum} &::= \mathit{product} + \mathit{sum} \mid \mathit{product} \\ \mathit{product} &::= \mathit{term} * \mathit{product} \mid \mathit{term} \\ \mathit{term} &::= \mathit{Number} \mid (\ \mathit{sum}\ ) \end{aligned}$$

Figure 2.1: Arithmetic Expression Grammar

This grammar is for basic arithmetic using only addition and multiplication. The non–

terminals, *sum*, *product* and *term*, are defined by their respective rules. Terminals *Number*, "+", "∗", "(", and ")" are implicitly defined with the exception of *Number*, which can be defined by the regular expression "[0..9]+". The importance here is for the scanner to accept a 3 as well as 647 as the same terminal, *Number*. The other four terminals, "+", "∗", "(", and ")" are discovered and returned to the parser as an acknowledgment of discovery for checking proper syntax. As an example consider the previous input, the string in 2.1, which contains white space that will also be discovered. The default operation is usually to ignore whitespace, but that is dependent on the language being developed and how the designer would like to handle it. The parser gets the identified strings along with their values, the number 9 or 647, and attempts to match the strings to the grammar that it is designed to recognize. Parsing of these identified strings is left to another part of the compiler [1, page 192].

2.2 Parsing

After the scanner has identified each component of a sentence and chopped the parts into "tokens" the parser then attempts to match the symbols supplied to the grammar. There are several parsing schemes. Which one is used is often dependent on how the grammar was written. LR parsing will be explained here since the integrated query language was implemented using LR; however, there are other relevant technologies such as LL(1), LL(k) and LALR [1, chpt 4].

LR parsing, often called bottom–up parsing, recognizes a sentence by taking the "right–most" derivation (the R in LR) of a rule to create a syntax tree [1, page 234], the end result of parsing. For the example in Figure 2.1 the start rule is on line 1. From this point the parser reads in tokens sent to it from the scanner and attempts to match them to the grammar by following all rules until either no rules match the input, resulting in an error, or a rule is completed, resulting in a reduction of the input [1, page 236]. Since LR is bottom–up the start rule is in reality where the parsing ends, not starts. The reduction

of input is in reality the popping off of tokens from a stack, where they were pushed upon recognition by a scanner, when a rule is completed. In the event of multiple rules being able to be completed the grammar would contain a reduce/reduce error which is solved by grammar rewriting in most cases. As an example, the string in 2.1 will be used again to show what the tree produced by the grammar in Figure 2.1 would look like. Figure 2.2 shows the syntax tree generated.

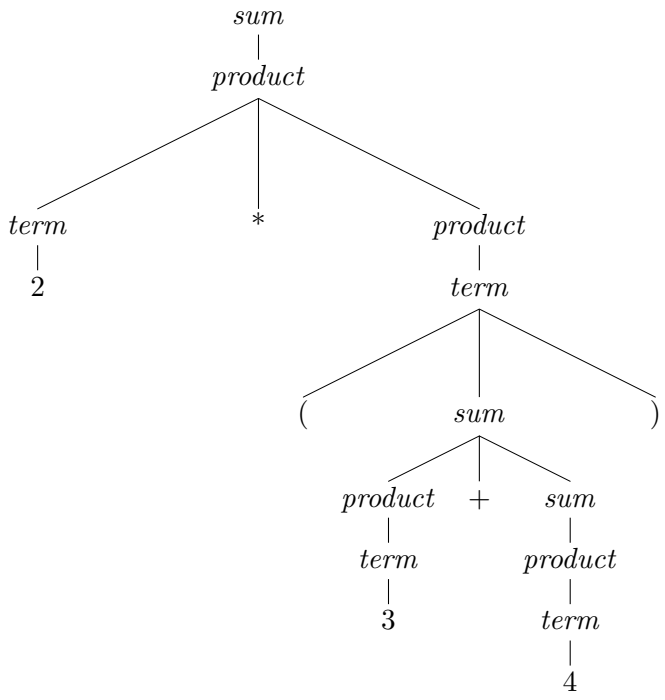


Figure 2.2: Syntax tree of string 2.1

This particular grammar is unambiguous, meaning that for every valid input there is only one tree that can be produced by the parser. In the event multiple trees could be generated the grammar would be deemed ambiguous and unfit for analysis [1, page 47]. A typical

$$S ::= S \text{ - } S \mid \boldsymbol{a}$$

Figure 2.3: Ambiguous Grammar

Figure 2.4: Parse trees for equation 2.2

ambiguous grammar is shown in Figure 2.3. It is quite easy to see that the string, in 2.2,

$$a - a - a \tag{2.2}$$

could be parsed in two distinct ways, by the grammar in Figure 2.3. If the first "a" is taken to be itself from the grammar, then the parser would consider the second part of the string "$a - a$" as the other S, which would subsequently reduce. The parser could also take the first part as "$a - a$" and the second string as "a", thus deriving another tree. In Figure 2.4 the two parse trees for equation 2.2 can be seen. This is what is meant by an ambiguous grammar, it is ill–suited for communication of *meanings*. Both trees are valid syntactically, but the meaning they possess is likely to be distinct ergo the programmer can not be sure the meaning of what they are writing and therefore communication between computer and programmer is non–existent. Meaning in a syntax tree, for illustration purposes, can be best represented in the example in string 2.1. The syntax tree is designed nested to ensure that the evaluation of the addition is performed before the multiplication, following the order of operations defined by mathematical convention. One can see then that the syntax tree generated is important not only in acceptance of source text, but in conveying meaning.

The meaning of the parsed text is still yet to be determined. The parser just ensures that the syntax of the sentence is correct and valid in the language. The analysis of this tree is when the ideas that are being expressed by the programmer are converted into something that can be mechanically acted upon [1, page 357].

2.3 Analysis

Analyzing the syntax tree created by a parse can be done in several ways depending on the needs of the programmer. A simple step by step evaluation of the tree is considered the ideal solution for an interpreter. In other instances running an interpreter may not be the ideal solution. For example, when memory is limited for running a separate environment or the programmer wants to run their code directly on the hardware, a stand alone program is desirable. Compilation of the input files to allow for execution on a separate machine are the more common form and require a target language, machine code in the case of C and byte code in the case of Java.

The analysis of this code can be as complex as the language designer wants. Decisions such as typing, functions, strict or non–strict evaluation, and memory referencing all play key roles in how the language is analyzed. Further considerations are architectural decisions where analysis could adopt a Visitor, Observer, or some other pattern, and how that implementation should be structured by using either object oriented programming or aspect oriented programming or both, in the tree analysis.

In a general sense the analysis of the syntax tree results in the concepts and ideas being converted to the target language, which the computer or underlying environment understands. The type of trees built by the parser impact the ease by which the analyzer can convert the source language to the target language.

Chapter 3

Conceptual Foundation

The system implemented is based on the Microsoft .NET^{TM} version 3.0 mechanism, LINQ (Language INtegrated Query)[2] whose underlying logic seems to owe much to the functional programming paradigm[8]. The basics of LINQ and some functional programming concepts will be discussed in this chapter to give the basis of the work done.

LINQ works by mapping an embedded query language to a syntactically correct host language. LINQ utilizes some features of the host languages, lambda expressions and type inference, allowing for a simplified mapping of operations on those data structures being queried. LINQ is currently designed to run in C# or Visual Basic. For the remainder of this thesis, examples will be in C#.

The LINQ design comes from the idea that every query operation can be translated into method calls on a query–able data structure[2]. For example, a programmer has a collection of objects and would like to perform some operation on each object. The current solution to this problem would involve a control structure such as a *for* loop, but this requires a lot of code and, given another similar task that code would have to be duplicated. However, a series of method calls on the collection as a whole would allow the programmer to accurately define the operation. The future programmer would then not have to just read code, but the method calls would have a defined API which could be referenced where the logic would

be explained.

3.1 LINQ Design

The programmer interacts primarily with the language side of LINQ. This consists of words that define a query on an object. Words such as **from**, **join**, **where**, and **select**, should bring SQL to mind when using LINQ. These words are used for their simple description of the action being performed. This abstraction continues in the method calls mentioned above. In much the same way that the method call `toString()` describes to the programmer that the object being called will be converted to a string, the methods that can be called in LINQ are identified in the same way.

For example, the programmer would like to select some elements of a collection. Intuitively, a method called `select()` would ideally perform this action. Therefore it is simple to see that `C.select(` *X* `)` would select from `C`, a collection, elements or extract from elements based on the supplied argument X (consider X to be some valid argument supplied to the method). The words mentioned above can then be seen as simple method calls on a collection. To illustrate, Figure 3.1, depicts the LINQ language and the subsequent translation as a series of method calls.

```
from c in C  //Define collection
join X       //Join collections
where Y      //Reduce collections
select Z     //Select from collections
```

```
C.join( X )     //Define and Join collections
 .where( Y )    //Reduce collections
 .select( Z )   //Select from collections
```

Figure 3.1: LINQ to Method calls

LINQ's main purpose is to map a language into a series of method calls in the host language. The host language then executes those methods calls and returns a new collection made up of elements representing the queried data. The queried data result utilizes a powerful tool, known as type inference[2]. This is a part of C# and allows the type of the result to be determined after the operation has been performed.

3.2 Type Inference

In LINQ, the programmer may not be able to predict or want to define the result of a query, which proves to be a problem in a strongly typed language like Java. C# however, has solved this problem with type inference. The result variable's type can be implicitly defined and delayed till evaluation. The presence of lambda expressions also utilizes type inference, by being able to avoid function typing prior to evaluation. The syntax for this feature is as follows:

```
var result = ... ;
```

This allows the *result* variable to be assigned a type after the expression on the right side of the assignment is evaluated[8]. LINQ uses this functionality to initialize the variable that will store the result of a query.

3.3 Extension Methods

The internal structure and functionality of a class is set once it is written and compiled. Extension methods in C# however allow the class to be extended with new methods that can operate on instances of that class. This is how LINQ is expanded on to aggregate classes that can now perform **where** and **join** logic. The description of such methods is very simple and can be done for any class.

```
public static class ExtensionMethods {
 public static char charAt(this string src,int index) {
  if (index<0||index>=src.Length)
    throw new IndexOutOfRangeException();
  char[] str = src.ToCharArray();
  return str[index];
 };
}
```

Figure 3.2: Extension method definition in C#

The C# code, in Figure 3.2, when inserted into a proper class, will allow for objects of type `string` to be able to execute the method `charAt()`, but this is just syntactic sugar for what is really going on. In reality the call on the source object is really a call on the stateless `ExtensionMethods` class with the object being called used as the first argument, as illustrated below.

`stringObj.charAt(1)` $\Rightarrow$ `ExtensionMethods.charAt(stringObj,1)`

Through the use of the extension methods mechanism, XML, Collection, and Database type objects now are fully extended with `where()` and `join()` methods to be used in LINQ queries.

3.4 Lambda Expressions

Data structures can be extended to have new logic and the result type declaration of an expression is delayed until after the expression is evaluated. Logic on how to evaluate each element of a collection is what is now needed. The lambda expression, a functional programming construct, provides this mechanism.

Lambda expressions are functions that are higher–order, meaning they can be used as an argument and/or returned as a result in the same way a scalar would be returned[8, 5]. The former is what allows LINQ to use element–manipulating expressions as arguments. Variables *X*, *Y*, *Z*, shown in Figure 3.1 represent logic that will be converted into lambda expressions. The C# syntax for lambdas follows lambda calculus syntax very closely and is shown below. In cases of implicit argument declarations the types are simply not included, in the argument declaration.

Explicit typing : $(type_1\ id_1, \ldots, type_n\ id_n) => body$

Implicit typing : $id_1, \ldots, id_n => body$

First–order functions now allow programmer–defined logic to be inserted into a method as

an argument. LINQ uses this to decide how it should evaluate each element of a collection. As an example, let's assume that the method `where()` has already been defined for the collection type `List`. Let us also assume that there exists a `List` object, `listOfStrings`. By supplying a lambda expression to the `where()` method the programmer can define what to do with each element in the list.

Explicit typing : `listOfStrings.where((string x) => x.Length > 10)`

Implicit typing : `listOfStrings.where(x => x.Length > 10)`

The above expression will return each element that has a length of greater than 10. The internal logic of the `where()` method is written to use the lambda argument. Note the result type of each lambda is boolean according to the definition of the `where()` method. The lambda itself does not declare the result type.

This feature along with those above provide the tools for LINQ to work. The syntax that was borrowed to make the Java–based querying language in this thesis is syntactic sugar for the mechanisms described above.

Chapter 4

Query Enhanced Language (QuEL)

This chapter describes a solution to query data structures within the Java language. Query Enhanced Language (QuEL) is a preprocessor that converts a Java code file containing embedded code into compilable Java. The programmer is responsible for preparing the collections as well as following the guidelines and respecting the limitations outlined in this thesis, in §4.2 and §4.4, respectively.

For purposes of studying the potential integration of the language into Java, packaging as a preprocessor was chosen in contrast to fully integrating the language, which would require development directly within the Java compiler, `javac`. The latter is of course advantageous from a programmer perspective and could be done with the open sourcing of Java[13] and the JVM[1]. This however could be premature as there are inherent differences between C# and Java, as outlined in chapter 3. Since Java does not possess these same features, this proof of concept was done.

4.1 Specification

QuEL is strongly based on LINQ and takes much of its structure from the LINQ proposal on the C# language [8]. The differences exist in the integration of QuEL in Java versus

[1] Java Virtual Machine

```
    ...
1)  @@
2)  over30 =  from Person s in staff
3)            where s.age() > 30
4)            select s.name()
5)  @@
    ...
```

Figure 4.1: QuEL example A

LINQ in C#. QuEL's grammar is defined in appendix A of this thesis.

4.1.1 Identifying

The QuEL grammar includes the embedding marker, @@. A pair of these markers tells the preprocessor that a QuEL query is contained within. The entire expression must be enclosed in these markers so the type of the result variable can be computed by the preprocessor, as would be done by the type inference mechanism in C#. The result variable of the query must adhere to the Java language specification for variable names. The type of this variable will be generated for the programmer, so no declaration is necessary. The query should be embedded within completely valid Java, since the preprocessor will not manipulate any part of the source code other than what is between the markers.

4.1.2 QuEL Examples

The code snippets contained in Figures 4.1 and 4.2 are pure QuEL. The ellipses represent valid Java code. In Figure 4.1 the `staff` in line 2 represents a collection that implements the `Iterable` interface and has been wrapped inside of the provided class `IterableImpl<X>`. All object collections must adhere to this specification. The internal collection in `staff` contains objects of type `Person` according to the query, line 2 in Figure 4.1 and 4.2. This is assumed to be correct and supplied by the programmer. QuEL requires that the programmer define the element type of the collections.

The query displayed in Figure 4.1 depicts the use of QuEL to find all of the objects con-

```
    ...
1)  @@
2)  bestRooms =  from Person s in staff
3)               join Room r in rooms on s.room matches r.room
4)               where s.tenure()
5)               where r.size > 300
6)               select new { name = s.name(), window = r.window }
7)  @@
    ...
```

Figure 4.2: QuEL example B

tained in the `staff` collection, line 2, whose `age()` method returns a value of greater than 30, line 3. The returned collection contains the values returned from the `name()` methods of those objects satisfying the previously described predicate, line 4.

The query in Figure 4.2 is more complex. Once again collection `staff` is being queried but the collection `rooms` is also included in this query. The query forces the joining of the two collections based on the room field of the `Person` objects in `staff` with the room field of the `Room` objects in `rooms`, line 3. The joining operation causes a product expansion of data based on the aforementioned fields. The two predicates on line 4 and 5 reduce the size of the collection while line 6 asks that the new collection make a new anonymous class object that contains the value of the `name()` method and `window` field from the `Person` object and the window field from a `Room` object. The return type in this instance is a collection of new anonymous type objects that have two fields, `name` and `window`.

4.1.3 Types

The typing system requires the element types of the collections used within a QuEL query to be compiled before the class containing the query is preprocessed. Reasoning for this is based on how the current preprocessor determines result types. This then also prohibits programmers from defining inner classes as the types of elements of collections within a QuEL query. Classes such as `Person` and `Room`, which are element types defined on lines 2 and 3 would have to be compiled prior to a QuEL query being processed.

Any type declaration that is used *must* also be specified in the fully qualified form (*i.e.* `java.lang.String` not `String`). Line 11 of Figure 4.3 illustrates this point. The elements types of the collections on lines 8 and 9 are not in packages, therefore, their type is correct. This format is a result of the method `forName()` in the class `Class` that is being used to get the `Class` object for that class type. The programmer should be aware that the code generated by QuEL may include inner classes, which are element types of intermediate collections or the resulting collection. These typing stipulations are based on the use of reflection in determining the return types on method and field references as well as constructor calls for new element generation.

4.2 Usage

QuEL is designed with an ease of query writing in mind. The syntax and style are similar to that of SQL and thus allow for an intuitive approach to selecting exactly what the programmer wants. The complexity of the query can range from a field of each element being selected to complicated joining of multiple collections and generation of new aggregate classes in the processes. QuEL makes no guarantees on the semantic correctness of the processed code, only that it is syntactically correct. The semantics of the query are dependent on how the programmer uses the query language. This will become evident as the programmer develops his/her skills at query design. For purposes of this section, the code contained in Figure 4.3 will be referenced.

The programmer must initially wrap the data structure to be queried in a QuEL data structure, as shown on lines 5 and 6 of Figure 4.3. Currently an interface for the Collections framework, `IterableImpl<X>` has been developed and must be imported into the class for use, see line 1 in Figure 4.3. In the referenced figure the entire package is imported as there are several classes that are needed. In the future if there exist other back–ends for different

```
1 ) import edu.rit.cs.quel.backend.*;
2 ) import java.util.ArrayList;
3 ) public class Query {
4 )   public void query (ArrayList<Person> _staff,ArrayList<Room> _rooms) {
5 )     final IterableImpl<Person> staff = new IterableImpl<Person>(_staff);
6 )     final IterableImpl<Room> rooms = new IterableImpl<Room>(_rooms);
7 )     @@
8 )     result = from Person s in staff
9 )              join Room r in rooms s.room() matches r.number()
10)              where s.age() > 25
11)              select new java.lang.String { s.name() }
12)     @@
13)     for (Object o :  result)
14)       System.out.println(o.toString());
15)   }
16)   public static void main(String[] args) {
17)     Query q = new Query();
18)     ArrayList<Person> staff = new ArrayList<Person>();
19)     staff.add(new Person("Anurag",29,100,true));
20)     staff.add(new Person("Aaron",26,100,true));
21)     staff.add(new Person("Gaurav",24,200,true));
22)     staff.add(new Person("Josh",25,200,true));
23)     staff.add(new Person("Manju",27,100,true));
24)     staff.add(new Person("Abhimanyu",21,200,true));
25)     ArrayList<Room> rooms = new ArrayList<Room>();
26)     rooms.add(new Room(100,2000,5551211));
27)     rooms.add(new Room(200,3000,5551212));
28)     q.query(staff,rooms);
29)   }
30) }
```

Figure 4.3: Full class example using QuEL

data structures those must also be wrapped so that QuEL may operate on them[2].

Figure 4.4 illustrates a simple query, which calls the `name()` method on each element of the `staff` collection and returns that result. The QuEL query is nested by the **@@** marker, as described in §4.1.1. The returned collection element type of this query is computed by the typing system, based on the **select** clause. In this particular example, the method call performed on the current element will return a `String` object, resulting in a collection

[2]Anurag Naidu is currently working on a XML and a database back–end[10]

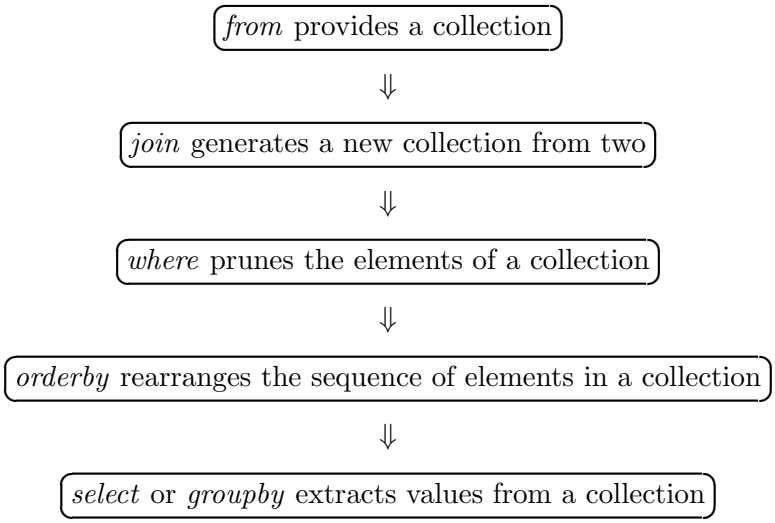
```
    ...
1)  @@
2)  result = from Person s in staff
3)           select s.name()
4)  @@
    ...
```

The result of this query would be a collection of strings with the following elements: `result` = [“Anurag”, “Aaron”, “Gaurav”, “Josh”, “Manju”, “Abhimanyu”]

Figure 4.4: Calling a method on each element of a collection

of `String` objects being assigned to the `result` variable. Line 11, shows the programmer defined the selected type with the fully qualified name, as defined in §4.1.3.

Queries must follow a given structure in QuEL. The design is enforced to formulate a type of pipeline that moves the results of previous query operations onto further operations. The pipeline can be seen in the below flow chart describing the operations of the clauses.



The sections below will explain most of the syntax and semantics of the language. For full syntactic reference, please refer to Appendix A.

4.2.1 From clause

In order to start a QuEL query or bring a collection into a query the programmer will use the **from** keyword.

from *type variable* in *collection* ...

The collection is identified, a variable is defined and typed which will represent each element of the collection. The type declaration is very important with the use of a preprocessor. This allows QuEL to reflect on the proper type for method and field calls.

4.2.2 Join clause

Following a *from* clause a programmer may wish to join a collection with another collection resulting in the product of the two. The syntax is much the same as the *from* clause at first glance, but along with the declaration of a collection, it's element type and a variable to use, there also must be a predicate that will function as a way to associate the two collections, the **on** and **matches** keywords serve this purpose.

... join *type* $variable_2$ in $collection_2$ on $expression_1$ matches $expression_2$ ...

The $expression_1$ variable represents a method call or field reference on a previously defined element of a collection. The $expression_2$ variable is what will be compared to $expression_1$. In the example in Figure 4.3, line 9 illustrates that all elements in the staff collection will be compared to all elements in the rooms collection by calling the `room()` and `number()` methods on the element variables of their respective collections. If the two expressions are equal, those elements are associated with one another in an intermediate tuple collection. Subsequent queries will refer to associated tuples.

If the programmer were to add the **into** keyword to the end of the *join* clause, the result

would be a "GroupJoin" which operates similarly, but groups the collections in a different manner. GroupJoins are different in that each element of the first collection would have with it a collection of elements from the second associated with it. The programmer supplies a variable that allows them to be referred to within collection elements in subsequent parts of the query.

... join *type* $variable_2$ in $collection_2$ on $expression_1$ matches $expression_2$ into $variable_3$ ...

The result type of this *join* clause would be as follows. Lets say $collection_1$ uses $variable_1$ to refer to its elements of $type_1$ and $collection_2$ using $variable_2$ to refer to its $type_2$ elements, the new element type would have the following structure.

`GroupJoin_Type`
- $type_1$ $variable_1$
- `IterableImpl<`$type_2$`>` $variable_3$

This element type contains one element from $collection_1$ and a collection where each element from $collection_2$ satisfies the "**on** $expression_1$ **matches** $expression_2$" predicate. The collection of elements from $collection_2$ satisfying the predicate can now be accessed via $variable_3$.

4.2.3 Where clause

Reduction of a collection is performed by the *where* clause. It simply takes an expression which when evaluated returns a boolean value. In line 10 of Figure 4.3, the expression will be performed on each of the elements of the collection associated with the variables in the expression. When the result of the expression is true, the element is included in subsequent queries.

... where *predicate* ...

Statement 4.1 is an example that will check whether a call to an elements `foo()` method

results in the value 0. If the predicate evaluates to true, the element is not pruned from the collection.

... `where` *elem*`.foo() == 0` ... (4.1)

4.2.4 Let clause

The *let* clause is similar to the keyword contained in the Scheme and Lisp programming languages. It allows the programmer to introduce a variable that is bound to a selection or smaller collection that can be subequently referenced in the query. In Figure 4.3, line 11 illustrates how a programmer could define the return type, along with what should be passed to the element type constructor.

... let *variable* = *expression* ...

... let *variable* = new { $expression_1$, ... , $expression_n$ } ...

Another example of the *let* clause, is in statement 4.2 that will create a variable, `n`, that contains the result of the right hand side of the expression for each element of the collection. If, for example, the `foo()` method returned a value of type `Integer`, the elements referred to by `n` would be computed to be of type `Integer` and could be referred to as such in subsequent queries.

... `let n =` *elem*`.foo()` ... (4.2)

4.2.5 Orderby clause

The order of the elements in a collection can defined by using the *orderby* clause. The programmer can order the the elements of a collection in ascending, the default, or descending order by using the **orderby** keyword. The direction of an order can be specified by using the **ascending** or **descending** keywords after the expression. Multiple orderings are also possible, with the expressions being comma separated.

... orderby *expression* ...

... orderby $expression_1$ descending, $expression_2$, ... , $expression_n$ ...

With multiple ordering expressions, the sequence of application can be thought of as a pipeline. The result of the application of first ordering is sent to the next and so on. Internally this operation is done through the use of the `Comparator` and `Comparable` interfaces. The *expression* is seen as the expression to sort each element on. Statement 4.3 depicts a small example.

... `orderby` *elem*`.name()` ... (4.3)

The result of the the applied ordering on

["Anurag", "Aaron", "Gaurav", "Josh", "Manju", "Abhimanyu"]

would result in the collection being sorted thus,

["Aaron", "Abhimanyu", "Anurag", "Gaurav", "Josh", "Manju"].

In contrast the sorted list would be reversed if the **descending** keyword was appended to the *orderby* clause

4.2.6 Select clause

The *select* clause completes the query pipeline by taking elements that have been created and or have passed all the predicates defined and returning them as elements of a new collection. The programmer may however only want part of these elements or may want to perform some operation on them, resulting in some type of specific data generated from each element.

... select *expression*

Generating collections of new objects from multiple queried collections can be done with the use of the **new** keyword. If the programmer has a type whose constructor takes the result types of the expressions, that can be called by defining that type, in fully qualified

form, immediately after the **new** keyword. This will then pass the results of the expressions as the arguments to the constructor of the type supplied. It should be noted that primitives will not work in this setting. Classes must be the argument types for the constructor of the type supplied.

... select new { $expression_1$, ... , $expression_n$ }

... select new *type* { $expression_1$, ... , $expression_n$ }

4.2.7 GroupBy clause

Similar to the *select* clause is the *groupby* clause. Here QuEL takes the elements of a collection and groups those elements by the expression supplied. The result is a new collection with elements of type `edu.rit.cs.quel.backend.Group`, an internal type in QuEL. This supplied type contains within it all the elements that evaluate the *expression* to the same result. The same result is defined by the returned boolean from the result types `equals()` method.

... group *variable* by *expression*

4.2.8 Query continuation

When a query has completed with a *groupby* or *select* clause it can then be passed into a new query. The purpose here is that some collections may need to be reduced or joined before being used in other queries. QuEL offers this functionality through the use of the **into** keyword. To insert the result of a *select* or *groupby* clause into a new query, simply define the variable which will refer to each element of the result collection and continue with the query. The type will have already been computed.

... group *variable* by *expression* into *variable QuEL_query* ...

... select *expression* into *variable QuEL_query* ...

In Figure 4.5, the result of the previous query, defined on lines 2 and 3, is being used and

referenced by the variable, `nms`, for the query continuation on lines 4 and 5. The entire previous query can be thought of as the initial *from* clause, therefore the programmer can continue as if they had just written it out. The result of this initial query is to select the value returned by the `name()` on each of the elements in `staff`, line 3. We will assume that the returned value is of type `String` for this example. The resulting collection is therefore a collection of `String` objects, whose elements are bound to the variable `nms`. Line 4 shows that the `length` field is being referred to. This is only possible because the variable is referring to elements of a collection whose type is `String`.

```
    ...
1)  @@
2)  result =  from Person s in staff
3)            select s.name() into nms
4)            where nms.length > 5
5)            select nms
6)  @@
    ...
```

Figure 4.5: QuEL continuation example

Refer to Appendix A for the precise grammar defining a query. Remember that the QuEL preprocessor merely ensures that the resulting generated code will be syntactically correct. Semantics of the query are left up to the programmer and their use of QuEL.

4.3 Architecture

The idea of QuEL and likewise LINQ [2] is that of a functional approach, which has recently been introduced in other programming language endeavors to help the programmer develop and represent complex interactions in an efficient way [9]. Functional programming concepts are essential in the QuEL architecture. It incorporates the idea of function composition by passing the result of one function to another as input. This should not be confused with the actual front–end logic for converting the QuEL language into compilable Java. QuEL is merely the preprocessor that converts the query language into a series of method calls on a collection similar to LINQ, §3. The back–end class, which wraps the object collections,

was written by Axel T. Schreiner[12]. This back–end will not be discussed.

The QuEL implementation utilized many features of Java. The following are some used areas of the language with a brief description of what areas were most helpful or hurtful by the feature.

Reflection An indispensable part of the QuEL architecture. Used heavily to determine how to build anonymous types and how to build the *method tree*

Generics Although helpful at times, proved to be more of a hurdle because of the issue of erasure[4]. This was used mostly in the back–end developed by Axel T Schreiner[12].

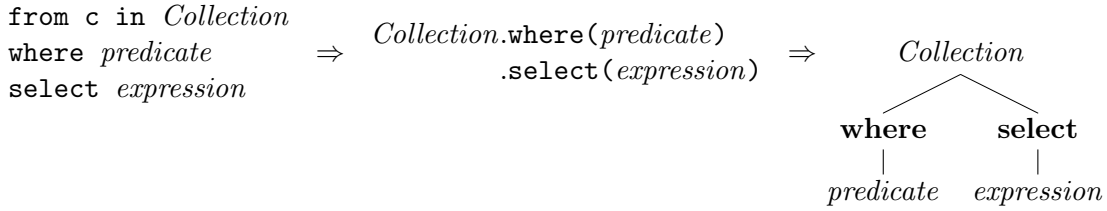


Figure 4.6: Query to Method tree

The logic of the QuEL implementation is to generate a syntax tree that can easily be converted to a series of methods calls that evaluate to arguments for further method calls[3]. The syntax tree is then converted to what is now termed a *method tree*. The *method tree*,shown in Figure 4.6, can be seen as a tree that any imperative language such as C or Java would create from a series of function calls. The root node is seen as the collection, being operated on. The children of the root are the query methods. Those query methods also have children, which are the arguments to those method calls. The current algorithm for conversion from QuEL syntax tree to the method tree is shown in Figure 4.7.

All nodes that represent a query action in the syntax tree have an equivalent node in the method tree. The node performs that action on the collection in the form of a method call. For instance, the `Where` node in the syntax tree is also a `Where` node in the method

[3]Since "method" is the Java term for "function" the two will be used interchangeably

Read in QuEL Java code
Parse QuEL code, generating syntax tree
Identify initial collection and element type
Process method invocations on collection:
 -Process subsequent *from*, *join*, *let*, and *where* clauses
 -- If *from* clause, begin processing methods on new collection
 -Process *orderby* clauses
 -Process Select or *groupby* clauses
 -If query is continued reprocess method invocations
Replace QuEL syntax tree segment with method tree

Figure 4.7: Conversion Algorithm

tree. This mapping of logical statements to method calls is the very backbone of the QuEL architecture. The query can be processed in one pass since the QuEL language is a translation into a series of method calls on a collection. The right side of Figure 4.8, illustrates the `where()` method call on a collection of `Person` objects. The translation scheme described above is exactly the way in which these methods will be called on the collection.

4.3.1 Functionality Extension

The methods described above are those whose inclusion in C# are defined by the extension methods described in §3.3. They allow the class to have extended functionality. In Java the implementation of these is not as direct. In C# a programmer defines a stateless class, Figure 3.2, which can then be used to define new methods for a class. Java does not have the syntactic sugar which allows for the neat method calls that C# uses when implementing extension methods. The resulting code is in fact the same for both C# and Java, but the intermediate step in C# is that the syntactic sugar is converted into method calls on a stateless class with the calling class being the first argument. Below is an example illustrating what C# is doing.

$$\texttt{obj.method(}\textit{arg}\texttt{)} \Rightarrow \texttt{StatelessClass.method(obj,}\textit{arg}\texttt{)}$$

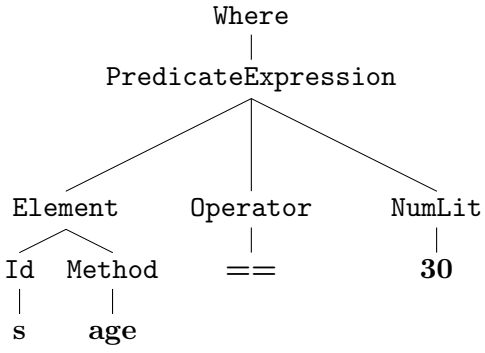
The right side of the equation is currently possible in Java but makes QuEL to Java translation harder and does not prove that QuEL can be done any better. The original solution,

proposed by Axel T Schreiner, involved exactly this. The solution defined a static class containing the appropriate methods, `join()`, `where()`, `select()` and so on, but made the resulting Java code terribly complicated and the translation was not as straight forward. The second implementation, the one currently used in QuEL, uses a type of wrapper class. This class extends and takes a class implementing the `Iterator` interface and allows the appropriate methods to be performed. The result is shown below.

```
Wrapper wrappedObj = new Wrapper(obj);
wrapperObj.method(arg);
```

This "wrapper" class is the back–end of QuEL. This wrapper needs to be written for each type of data structure QuEL will query. The designer need only define how methods such as `join()`, `where()`, `select()` and so on are performed on these classes.

The arguments to these method calls are what are known as closures, or lambda expressions. These lambdas are functions that have no name. They are similar to C# delegates. They must be *single* argument functions that perform an operation on the argument and return the result. For instance, in the **where** logic the predicate makes up the body of the lambda with an element of the collection being the argument.




```
...  .where(new Func {
 public Boolean f ( Person s ) {
  return s.age() == 30;
 }
}).  ...
```

Figure 4.8: Method Tree and Compilable Java for `where s.age()==30`

4.3.2 Currying

There does exist a special method call used when bringing in multiple collections. The algorithm specifies that at some point another *from* clause can be declared that would bring in a new collection. It is also known that QuEL results in a series of method invocations on a wrapped collection. The problem presents itself that a new collection must also be iterated over at the same time as the initial collection and must result in a collection that includes elements from this new collection.

This is exactly where lambda expressions present a problem. Lambda expressions have are defined to take one argument only. Equation (4.4) shows how a simple function is defined in traditional lambda calculus notation. The problem is that each method takes a function that operates on each element of a collection one at a time. This is not feasible if the programmer is iterating over multiple collections and there will be multiple arguments present in the query.

$$(\lambda\ arg\ =>\ body) \tag{4.4}$$

Currying or Schönfinkelisation is the process of taking a function and reducing the number of argument that it takes from many to one[11]. The theoretical study of this process is done in lambda calculus which also happens to be the basis for the lambda expressions that are being used here. The above problem is solved by the concept of currying (making a single argument function accept multiple arguments).

Thus far it is know that a query will look at each element of a collection and perform some operation on each one. It was also stated that each lambda expression, which is to be the argument for the query methods we are mapping to, take only one argument. This argument can be assumed to be a single element of a collection. There exists the potential for multiple collections to be present in a query without being joined. The below example shows that there can be multiple collection element referenced within a *where* clause.

```
from Person s in staff
from Room r in rooms
where s.roomNumber == r.number
select s.name
```

From the above example we can deduce some things described in this section and §4.2. First we know that the predicate following the *where* clause will be contained within a lambda expression and lambdas were previously defined to only take one variable, resulting in a lambda that looks like the following.

```
( λ r => s.roomNumber == r.number )
```

However, there are two variable contained within the predicate only one of which is bound, `r`. The `s` variable is free but needs to have some value that will make the predicate a valid expression. Equation 4.5 depicts how multiple lambdas with one argument can be nested to introduce many arguments into the body. This is how several collections are being brought into a QuEL query. The `SelectMany()` method takes a lambda that has a nested lambda as its argument.

$$(\lambda x \ => \ (\lambda y \ => \ x \ y)) \tag{4.5}$$

Applying 4.5 to the previous example we get what is actually supplied to the `SelectMany()` method. A curried function that takes one argument instead of the two that was previously needed.

```
( λ s => ( λ r => s.roomNumber == r.number ))
```

4.3.3 Transparent variables

The instantiation of anonymous types and aggregate class types for the dynamic selection and joining of elements respectively presents a problem at compile time. For example when a collection is brought into a query, the programmer must define the type and variable that will refer to elements of that collection. This is not so in instances where the programmer

or QuEL creates a new type. Java has no facility for adding fields to an element class let alone methods that can be called, so new types must be created.

New types are most often created during *select* clauses such as

```
... select new { name = s.name(), window = r.window }
```

from line 6 of Figure 4.2, in the last section. The code states that a new class will be created that contains 2 fields, `name` and `window`. Java has no facility for this operation, therefore to be done on the fly, the programmer would have to use something like a hashmap for storing the field name as the key and the evaluated expression, field or method call, as the value. This however does not allow for proper type checking, since the programmer would not be able to define the type of each entry in the hashmap. Dynamic class generation through the use of the QuEL preprocessor and reflection allow the programmer to have actual classes generated with proper fields and specific types. The result is a type–safe query on the fly.

Inclusion of collections into a query are done through the *from* clause which requires a type, variable, and collection identifier. If a new variable or field is added to elements of a collection or multiple collections are joined resulting in aggregate elements, the variables the programmer defined may no longer be valid, from the compiler's perspective. For example, a programmer could bring in two collections, A using a as the element variable and B using b as its element variable. If a programmer were to then **join** the two collections on key, it would be the product of the two collections, based on the key, as seen in equation 4.6.

$$
\begin{aligned}
A &= \begin{bmatrix} a_1 & a_2 & \cdots & a_m \end{bmatrix} \\
B &= \begin{bmatrix} b_1 & b_2 & \cdots & b_n \end{bmatrix} \\
C &= \{(a,b) \mid a \in A,\ b \in B,\ a.key = b.key\} = \\
&= \begin{bmatrix} c_{a_{key=i},b_{key=i}} & \cdots & c_{a_{key=j},b_{key=j}} \end{bmatrix}
\end{aligned}
\tag{4.6}
$$

The resulting collection C has a new set of elements made up of an aggregate type that was not explicitly supplied by the programmer. The collection can then be passed on to subsequent actions such as **where** or **select** logic; however, the collection will be made up of some new type, not a type defined by the programmer. Method calls or field references that the programmer would like to call on elements of collection A will then be written as `a.method()` or `a.field`, respectively, which does not mean anything to the elements of this new collection. The programmer would really like to reference the field of each element in C that contains the elements from A, `c.a.method()` or `c.a.field`.

The solution to this problem is to mark types in the query that have variables associated with them. QuEL prepends any collection elements that does not have a variable associated with its type, with the variable that is currently in the scope of the operation. It is not important to know the type, just that a variable has been defined, and is in scope. This is so because QuEL will not check the semantics of the query. If the programmer chooses to make a field or method call on a type that does not have that field or method, that is a semantic error that will have to be picked up later by the Java compiler. QuEL will only check the syntax and ensure that it has been translated into the appropriate Java code.

4.4 Limitations

The current QuEL implementation contains some limitations that hinder what one can do within a query as well as what can be done in the surrounding Java code. The limitations are

based on current difficulties with integrating the QuEL logic outside of the Java language. If QuEL were fully integrated into the Java language definition, all of these limitations would disappear. Below is a non–exhaustive list of limitations that one may encounter during usage. This list is non–exhaustive because of the infinitely complicated queries that database programmers could theoretically construct.

4.4.1 Parser

The parser itself does not actually parse Java code. It reads the file looking for a QuEL query and if found will convert it. A parser that recognizes proper Java code along with QuEL, would allow programmers to not be forced into using the fully qualified name of a class and could also read the parameterization that is used for Generics. The parameterization data would allow for the removal of all typing information from the query language.

4.4.2 Element types

Generic element types was a potential feature that was removed from the logic. It was removed to simplify the resulting parse tree translation. When the elements of a collection utilize Generics, as in `ArrayList`⟨`Node`⟨`Person`⟩⟩, the system will parse the element type correctly; however, the resulting query may not be syntactically or semantically correct and therefore not compilable. This is a result of erasure[4], the parameterized type not actually being read separately from the Generic class. What QuEL will use is reflection on the string "Node⟨Person⟩" which makes no sense to the reflection API. This translates into the system not understanding the type of the element, which means it is impossible to determine valid field and method references which may be made by the programmer in the query. Consequently, parameterized types cannot be used as elements of a queried data structure.

4.4.3 Expressions

QuEL uses predicate expressions for determining which elements it will select. Predicates must evaluate to type *boolean*, according to the `where()` method definition. Therefore the

logical operators are included. However, the arithmetic operators, such as + and − have been excluded from being a part of valid expressions. This was done since to simplify the preprocessor and maintain the focus on the development rather than the design of a trivial expression evaluator.

The operators that a predicate expression can contain are <, >, >=, <=, ! =, ==, &&, and ||. For example `n.age() + 1 < 8` is not valid; however, something like `n.age().add(1) < 8` would be valid, and accomplish the same result. This is of course dependent on `add()` being a valid method on the return type from the `age()` method.

4.4.4 Dynamic Casting

Since QuEL is designed for typing, the need for casting should be eliminated. This however is not the case in one particular instance. The result of the `groupJoin()` back–end method is an element type that contains a previously defined type and a `IterableImpl<X>` type, that contains a list of elements that are associated with the previously stated single element. Due to the back–end design however, the returned element type will not have the precise type of `IterableImpl<X>`. It will instead belong to the superclass `Iterable`, shown in Figure 4.9. The problem arises when a programmer would like to call a method on the list part of the resulting element type.

As an example the `IterableImpl<X>` contains a `size()` method that returns the number of elements the collection contains. If the programmer was looking to such a method call on the `list` field of the class (Figure 4.9), the method call would need to be defined on type `Iterable`, not `IterableImpl<X>`. Polymorphism allowed an element of type `IterableImpl<X>` to be used as the second argument for this class, since it is a subclass of `Iterable`, but its precise type is lost. Through the use of reflection on the back–end, this result type is determined.

Casting would need to be performed at this point but is not possible if a generic front–end

```
public class ResultElementClass {
   public type element;
   public Iterable list;
   public ResultElementClass( type element, Iterable list) {
      this.element = element; this.list = list;
   }
}
```

Figure 4.9: Example of `groupJoin()` result element type

is to be designed. A restructuring of the back–end may solve this problem, as not wrapping the collections would remove the casting, but lose the method call. One could also expand the preprocessor to make an elaborate checking system to perform a cast at the proper time. The latter solution was examined, but proved to be outside the scope of this thesis.

Chapter 5

Comparisons and Examples

QuEL has been shown to perform querying quickly and without complications resulting from multiple control structures and complicated logic. This section will expose some other solutions to the query problem, along with what the programmer would do with standard Java constructs. An obvious trend that can be observed in all of the alternative solutions is the fact that the individual language expressions are on the same level as Java expression. This means that they can be utilized as predicates in control structures or right hand side expressions for assignments. QuEL must be formatted in the described way, thus limiting its application to right hand assignment expressions.

5.1 Java without QuEL

The loop is the control structure that comes to mind when iterating over some collection. This is by far the most direct approach of performing a controlled deterministic way of looking through a collection. It is the intent here to show the programmer an example of how QuEL could significantly simplify a loop based query over a collection. The example in Figure 4.2 will be rewritten in Figure 5.2 as how it would be done in Java.

The result of the rewriting is a complicated set of nested control structures that work, but without heavy commenting a reader would be lost. It is not even obvious that the

```
...
@@
bestRooms =  from Person s in staff
             join Room r in rooms on s.room matches r.room
             where s.tenure()
             where r.size > 300
             select new { name = s.name(), window = r.window }
@@
...
```

Figure 5.1: QuEL query

two figures contain the same logic. Aside from the obviously convoluted code, the types were not checked at any of the steps. The storage of the resulting collection is in the form of a generalized `HashMap`. The information pertaining to what the resulting `HashMap` contains or how it can be retrieved is lost in generalizing the values in the resulting list elements.

```
List<Map<String,Object>> bestRooms = new Arraylist<HashMap<String,Object>>();
Map<String,Object> ne;
for (Person s : staff)
  for (Room r : rooms)
    if (s.room == r.room) {
      if (s.tenure() && r.size > 300) {
        ne = new HashMap<String,Object>();
        ne.put("name",s.name());
        ne.put("window",r.window);
        bestRooms.add(ne);
      }
    }
```

Figure 5.2: Figure 5.1 rewritten without QuEL

5.2 Saffron

Similar to SQL in form, a query solution called Saffron[7], last updated in 2004, offers an inline solution to query a collection. Like QuEL, Saffron uses a preprocessor, however it does have the added benefit of a query optimizer. A Saffron expression can replace a Java

expression in any position. The predicates in loops and other control statements can be replaced with Saffron expressions. Figure 5.3 shows the use of a Saffron expression as the predicate in an *if* control structure.

```
if (exists (select from emp where emp.salary > 50000)) { ... }
```

Figure 5.3: Saffron example from website[7]

Whereas QuEL is syntactic sugar for a series of method calls, Saffron is syntactic sugar for loops over collections. Saffron does not support a back–end option which would allow for alternative data structures to be queried, such as the query solutions being proposed in [10]. Typing is also a limitation of Saffron, in that it does not perform type checking. This is most obvious in that all returned types are aggregates that contain elements of type `Array`.

5.3 JQL (Java Querying Language)

JQL is a query mechanism specifically for collections[14]. There is no back–end for alternative data structures. Although that is a downside to QuEL, it can be fully embedded within Java without tags as are required in QuEL. JQL expressions are fully equivalent to Java expressions and the programmer can insert them any where a Java expression is valid, similar to Saffron as discussed in §5.2. The example in Figure 5.4 illustrates the use of JQL as a Java expression. A preprocessor still must go over this code and convert to compilable Java, however, tags are not needed.

```
List<Object[]> matches = selectAll(String w :  words,
                                   Integer i :  gaplengths |
                                   w.length() == i);
```

Figure 5.4: JQL example from website[15]

The Figure 5.4 example iterates over two collections comparing the length of each `String` in

the `words` collection with the `Integer` in the `gaplengths` collection and returning a list of `Object` arrays, each containing a `String` object and `Integer` object that satisfy the defined predicate. JQL, however does no type checking nor does it allow the programmer to define what they want returned.

The JQL solution does satisfy a niche for assertions when a simple expression can define exactly what the programmer would like to check. This was the original intention of JQL a subset of the Squirrel architecture[14]. JQL lacks the type checking that QuEL possesses, JQL only processes the right side of the expression, not the assignment.

Chapter 6

Conclusion

The whole of this thesis was to see if a tool like LINQ[2] could be implemented effectively within Java even though Java does not contain type inference (§3.2), extension methods (§3.3), or lambda expressions (§3.4). The remaining sections of this work deal with the answer to that question. Each section examines an area that could expand the work done here or allow QuEL to be integrated into the Java language. The final section discusses a new feature that is currently being proposed for Java and that could push QuEL into a true functional solution.

6.1 Future Work

This thesis presents a vast amount of possible future projects and modifications to the current ground work that has been done. Along with the limitations, §4.4, there is the development of multiple back–ends that could also be explored.

Two areas that are in LINQ that have not been explored by this paper are XML and Database data structures. Currently there is work being done on these two back–ends[10]; however, the specifics involved in typing as well as aggregate elements still needs to be explored. The parser for XML and Database queries may involve a more verbose expression language. The current implementation does not allow for arithmetic expressions; this could

be done at the same time. The lexer and parser solutions to these problems are trivial, but the typing of these resulting expressions would require more effort.

6.2 Java Closures

There is currently a proposal for closures, lambda expressions, in Java[3]. A Java with lambda expressions would allow a drastic simplification of the translation into compilable code. The proposal defines a syntax as well as an operating definition along with the impact on the current language design. The semantics of the proposal will not be discussed here, only what the resulting translations could look like as compared to the current QuEL implementation.

$$\{\ FormalParameters_{opt} => Statements_{opt}\ Expression_{opt}\ \}$$

Figure 6.1: Proposed closure syntax[3]

The syntax of the proposed closures is illustrated in Figure 6.1. This is a much cleaner syntax than the current QuEL solution of an anonymous class being instantiated. Figure 6.2 illustrates the current solution with what it would look like with closures.

```
...  .where(new Func {                    ...  .where(
 public Boolean f ( Person s ) {           { Person s => s.age() == 30 }
  return s.age() == 30;                   ).  ...
 }
}).  ...
```

Figure 6.2: QuEL without/with closures for `where s.age()==30`

The impact of this work proves that LINQ can be done within the Java language and the mechanisms, although not as simple and direct as in LINQ, do exist in Java. The impact of QuEL as a helpful addition to the Java language is yet to be determined and hypothesizing its importance would be a premature exercise. There is potential for impact if other language features maklike those now part of within C# are integrated within Java.

Appendix A

Language Grammar

This describes the structure of the QuEL language, written out in EBNF[1]. Bold or quoted words are terminals, italics indicate non–terminals. Regular expressions are used in some non–terminal definitions for precision.

unProcessedCode ::= *javaCode queryAssignment unProcessedCode* | *javaCode*

queryAssignment ::= **@@** *id* **=** *queryExpression* **@@**

queryExpression ::= *fromClause queryBody*

queryBody ::= *joinClause* fromLetWhere* orderbyClause*? *selectGroupby queryCont*?

fromLetWhere ::= *fromClause joinClause** | **let** *id* **=** *selExpr* | **where** *predExpr*

selectGroupby ::= **select** *selExpr* | **group** *id* **by** *keyExpr*

fromClause ::= **from** *type id* **in** *element*

joinClause ::= **join** *type id* **in** *element* **on** *element* **matches** *element into*?

into ::= **into** *id*

orderbyClause ::= **orderby** *orderbyExprList*

orderbyExprList ::= *orderbyExpr* / **,**

orderbyExpr ::= *element ascendOrDescend*?

ascendOrDescend ::= **ascending** | **descending**

queryCont ::= *into queryBody*

[1]Extended Backus Naur Form

anontypedeclare ::= *id* **=** *valExpr* | *element*

selExpr ::= **new** *type*? **{** *anontypedeclare* / **,** **}** | *element*

srcExpr ::= *element*

keyExpr ::= *element*

valExpr ::= *queryExpression* | *element*

predExpr ::= *element* / *operator*

element ::= *id memberRef*? | *literal*

memberRef ::= **.** *memberType* / **.**

memberType ::= *id method*?

method ::= **(** *arguments*? **)**

arguments ::= *element* / **,**

literal ::= *strLit* | *charLit* | *intLit* | *dblLit* | *boolLit*

operator ::= “<” | “>” | “>=” | “<=” | “! =” | “==” | “&&” | “||”

boolLit ::= **true** | **false**

strLit ::= Java string surrounded by double quotes

charLit ::= Java char surrounded by single quotes

intLit ::= [0-9]+

dblLit ::= ([0-9]*.[0-9]+) | ([0-9]+.[0-9]*)

id ::= [a-zA-Z]+[a-zA-Z_0-9]*

type ::= Class described using fully qualified path. *e.g.* `java.lang.String`

javaCode ::= Non–terminal representing valid Java code.

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Box, D., and Hejlsberg, A. The LINQ project .NET Language Integrated Query. http://www.microsoft.com/downloads/details.aspx?familyid=1e902c21-340c-4d13-9f04-70eb5e3dceea, May 2006.

[3] Bracha, G., Gafter, N., Gosling, J., and von der Ahé, P. Closures for the java programming language (v0.5). http://www.javac.info/closures-v05.html, February 2007.

[4] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)* (Vancouver, BC, 1998), C. Chambers, Ed., pp. 183–200.

[5] Church, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics 58* (1936), 354–363.

[6] Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, March 1979.

[7] Hyde, J. Saffron. http://perforce.eigenbase.org:8080/@md=d&cd=//open/saffron/doc/&c=gqc@//open/saffron/doc/index.html, June 2004.

[8] Microsoft® Corporation. C# version 3.0 specification. http://msdn2.microsoft.com/en-us/library/ms364047(vs.80).aspx, September 2005.

[9] Murthy, C. Advanced programming language design in enterprise software: a lambda-calculus theorist wanders into a datacenter. *SIGPLAN Not. 42*, 1 (2007), 263–264.

[10] Naidu, A. Language integrated query in Java for XML and relatonal database. Master's Thesis Proposal – Rocester Institute of Technology, May 2007.

[11] Schönfinkel, M. Über die bausteine der mathematischen logik. *Mathematische Annalen 92* (1936), 305–316.

[12] Schreiner, A. T. LinQ — why we have to teach functional programmming. Presented at Rochester Institute of Technology on December 5 as part of the 2006 colloquium series, December 2006.

[13] Sun® Microsystems Inc. Sun opens java. http://www.sun.com/2006-1113/feature/, November 2006.

[14] Willis, D., Pearce, D. J., and Noble, J. Squirrel : A query based debugger for java.

[15] Willis, D., Pearce, D. J., and Noble, J. JQL : The java query language. http://www.mcs.vuw.ac.nz/ darren/jql, 2006.