

JSINQ – A JavaScript implementation of LINQ to Objects

Kai Jäger
Hochschule der Medien, Stuttgart
Computer Science and Media (Master program)
Stuttgart, Germany
jaeger@xwrs.net

DRAFT

February 15th, 2009

Abstract

Much of the code in today's client and server applications deals with the aggregation of data from various data sources. While there are query languages for relational databases and XML documents, until recently there has not been a mainstream query language for in-memory collections. The Language Integrated Query (LINQ) feature available in C# and VisualBasic.NET addresses this issue and also provides unified querying capabilities for relational databases, XML documents and many other data sources. Unfortunately, LINQ has not found wide distribution outside of the .NET community and implementations of LINQ in other languages tend to be either incomplete or non-existent. This paper discusses JSINQ which was written to be the first publicly available complete implementation of LINQ to Objects in JavaScript.

1 An Introduction to LINQ

Many of today's business applications spend much of their time aggregating data from different heterogeneous data sources. Most commonly, these data sources are relational databases and XML documents. However, an application may also have several "internal data sources", that is in-memory collections filled with data that has been generated programmatically (usually by means of transforming data acquired from external data sources).

Data is usually retrieved from these data sources using some form of query language. For relational databases, that language is typically some dialect of SQL and for XML documents it is mostly XPath or XQuery. APIs for querying in-memory collections on the other hand are not very common in most mainstream languages, which is why queries against those kinds of data sources are often written by hand in an imperative style (i.e. using loops). Alternatively, some languages provide a small set of higher-order-functions (map, filter and fold) that can be used to write simple queries against those data sources as well.

There are two main flaws to be identified with the way we write queries today. For one, there is an impedance mismatch between the imperative programming languages we write our applications in and the declarative languages we use to query our databases or XML documents. When we embed a SQL query in a language like Java or C# for example, we typically enclose it in quotation marks to "hide" it from the compiler. That of course has the undesirable side-effect that we do not get any compile-time checks for our queries. Any type errors in our queries are deferred until they are executed and even syntax errors do not normally get caught by the compiler. The second problem with the way we write queries today is that there is no uniformity. While SQL and XQuery are reasonably similar in their syntax, they are not the same language and anyone who wishes to write queries against relational databases and XML documents will ultimately have to learn both. Additionally, while SQL is an ISO standard, there are in fact many SQL dialects, each enhanced with vendor-specific extensions. Finally, querying in-memory collections until recently has been an area, largely untapped by generic APIs or domain specific languages.

With the introduction of the .NET Framework 3.5, Microsoft has attempted to bridge the gap between relational databases, XML documents and imperative programming languages and has also introduced a new query language that is generic enough to work with a wide variety of data sources including relational databases, XML documents, in-memory collections, the Windows Registry, LDAP directories and many more.

1.1 LINQ in C# and VisualBasic.NET

Both C# and VisualBasic.NET now support query expressions as an integral part of their syntax. These query expressions are syntactically similar to SQL but because of their general-purpose nature, they lack any notion of databases or tables. Instead, LINQ queries always operate on objects. These objects may be the nodes of an XML document, the elements of an in-memory collection or the objects generated by an object-relational-mapping framework.

```
var resultSet = from customer in customers
                join order in order on customer.id equals order.customerId
                orderby order.date descending
                select order;

foreach (var row in resultSet)
{
    Console.WriteLine(row.quantity);
}
```

Listing 1: A simple LINQ query in C#

Listing 1 shows how query expressions can be integrated into C# code. Note that the query is not enclosed in quotation marks. Instead, both the compiler and the IDE are aware of the query and will provide meaningful error messages and even statement completion. Also note that the variables *customers* and *orders* used in the query above could refer to either an in-memory collection or a table in a relational database (through some kind of mapping mechanism).

1.2 LINQ behind the scenes

LINQ is a DSL for writing queries, but it is also a library. In fact, the query expression syntax used in Listing 1 is strictly a compile-time feature. There are no constructs in the CIL¹ to represent LINQ queries and the CLR² has no understanding of LINQ whatsoever. Instead, LINQ queries are translated into method calls by a pre-processor. The query in Listing 1 would be translated into the following:

```
customers.Join(
    orders,
    (customer) => customer.id,
    (order) => order.customerId,
    (order, customer) => new {order, customer}
).OrderBy((X) => X.order.date).
Select((X) => X.order);
```

Listing 2: A LINQ query written using extension methods

¹ Common Intermediate Language – The .NET bytecode format.

² Common Language Runtime – The .NET VM.

The *Join*, *OrderBy* and *Select* methods are part of a larger set of “standard query operators”. These operators can be defined explicitly or they can be “inherited” through an extension method mechanism³. Note that these methods are in fact higher-order-functions as they accept other functions (or lambda expressions) as their parameters⁴.

The “standard query operators” are extension methods that operate on objects that implement either the *IEnumerable*- or the *IQueryable* interface. The LINQ API used to write queries against in-memory collections (appropriately named “LINQ to Objects”) operates only on the *IEnumerable* interface, which is reproduced below.

```
public interface IEnumerable<T> : IEnumerable
{
    public IEnumerator<T> GetEnumerator();
}
```

Listing 3: The *IEnumerable* interface

```
public interface IEnumerator<T> : IDisposable, IEnumerator
{
    public T Current { get; }

    // Inherited from IEnumerator
    public bool MoveNext();
    public void Reset();
}
```

Listing 4: The *IEnumerator* interface

It should be apparent from the two listings above that *IEnumerable* is an implementation of the Iterator Pattern. With this in mind, the inner workings of LINQ are now easily explained:

If you look at Listing 2 again, you will realize that the extension method *Join* is invoked on the *customers* object. Then the *OrderBy* method is invoked on whatever the return value of *Join* is. Finally, the *Select* method is invoked on what is returned by *OrderBy*. This kind of method chaining is possible, because each of these methods returns an object that is an instance of the *IEnumerable* interface. Put differently, each query operator performs a transformation of an input that is an *IEnumerable* to an output that is also an *IEnumerable*⁵. Because input- and output-types of the query operators are symmetrical, composing

³ Extension methods are a way of adding new methods to existing classes. Unlike “open classes” which exist in dynamic languages like Ruby or Python, extension methods are resolved at compile-time and do not have a runtime overhead.

⁴ When writing queries against relational databases, the lambdas are actually translated into expression trees. This allows the OR-mapper to dynamically generate SQL statements from LINQ queries.

⁵ This is true for many but not all query operators. There are also those which return scalars (e.g. the *Sum*-or *Average*-methods).

different query operators is now rather trivial. Because C# and VisualBasic.NET respectively are object-oriented programming languages, this type of function composition can happen implicitly through dot-notation.

1.3 LINQ and JavaScript

There are several ways of accessing LINQ from JavaScript: the more direct way is to use JScript.NET which is Microsoft's JavaScript implementation for the CLR. Alternatively, one could use Volta, a tier-splitting framework that features a CIL to JavaScript compiler. Since JScript.NET cannot be used in the browser without additional plugins, this solution lacks practicality. Using Volta on the other hand is certainly feasible, but the framework is most commonly used to compile C# or VisualBasic.NET code to JavaScript, which in turn means that you would not actually be writing your LINQ queries in JavaScript directly.

These observations lead to the question if there even is a real need for a JavaScript implementation of LINQ and whether such an implementation would actually be useful. As JavaScript does not offer language support for LINQ, any implementation of LINQ in JavaScript would either have to be confined to its library part, be implemented as a pre-processor or require query expressions to be enclosed in quotation marks. The latter might contradict LINQ's primary goal of being "language integrated", but in the presence of an interpreted or dynamically compiled host language, a lack of compile-time verifiability essentially becomes a non-issue.

1.3.1 Previous work

In the past, there have been multiple attempts at implementing LINQ in JavaScript. Most notably there are [Pietschmann, 2008], [HBoss, 2008] and [Chandra, 2007]. While all of these implementations are both significant and useful, none of them implement the whole set of "standard query operators" found in LINQ and they also do not provide the means to write query expressions using the comprehension syntax that is now supported by C# and VisualBasic.NET.

JSINQ, which is the subject of this paper, was developed to be the first publicly available JavaScript implementation of LINQ to Objects that not only supports all "standard query operators" but also provides a runtime compiler that translates query expressions into JavaScript code.

2 Implementing JSINQ

2.1 Standard query operators

As explained in 1.2, LINQ is primarily a library. The first step in developing JSINQ therefore was to re-implement said library in JavaScript. For the most part, this turned out to be reasonably easy to do. There were however a few challenges in replicating LINQ's deferred execution behavior: when writing a querying against a collection for example, the query is not actually executed until the result set produced by the query is enumerated⁶. This behavior is useful, as it prevents the mere definition of a complex query from slowing down the application but it also enables the developer to write certain types of queries against very large or (in theory) even infinite data sets. More importantly though, using deferred execution behavior, a query can greatly reduce its need to keep intermediate results in memory.

```
customers.Where(  
    (customer) => customer.lastname.StartsWith("S")  
) .Select(  
    (customer) => customer.lastname  
);
```

Listing 5: A query with a Where-Clause

In the above example, the *Where*-operator is used to filter the *customers* collection so that only customers whose last names begin with the letter "S" are retained. A more conservative implementation of the *Where*-operator might iterate over the *customers* collection and then add each element that satisfies the before mentioned condition to a new intermediate collection. However, depending on the size of the *customers* collection, the intermediate collection might consume a lot of space. Also, the intermediate collection would never actually be accessed in its entirety, as the *Select*-operator that follows the *Where*-operator has to look at each input element individually. All "standard query operators" that operate on one element at a time are therefore implemented using deferred execution, i.e. they hold no intermediate state and they only retrieve new elements from their input when their output is itself enumerated.

In C# and VisualBasic.NET, this form of deferred execution is fairly easy to implement as both languages support generators. JavaScript on the other hand did not support generators until version 1.7, which unfortunately has not yet found wide acceptance. JSINQ there-

⁶ This is not true for queries containing *OrderBy*-, *GroupBy*-, *Join*- or *GroupJoin*-clauses.

fore implements deferred execution without generators using closures. This works well but results in a lot of boilerplate code that could otherwise be avoided.

2.2 Query expressions

In 1.2 we have established that the query expression syntax supported by C# and Visual-Basic.NET is in fact syntactic sugar and that all query expressions are translated into method calls by a pre-processor. In turn, this means that every query that can be expressed as a query expression can also be expressed as a sequence of method calls. In fact, because C# has a fairly compact notation for lambda expressions, queries written using the extension method syntax are sometimes shorter than those written using the more declarative query expression syntax. Unfortunately, the lambda expression syntax in JavaScript is a lot less compact and so JSINQ queries written using method calls tend to be rather long and difficult to read as illustrated in the following example:

```
customers.where(function(customer) {
    return customer.name.charAt(0) == 'S';
}).select(function(customer) {
    return customer.name;
});
```

Listing 6: A JSINQ query that uses the method syntax

While a slightly more concise syntax for lambda expressions has been introduced with JavaScript 1.8, it is not yet supported by the majority of JavaScript implementations.

The JSINQ query compiler was written to circumvent this issue. It receives a string containing a query expression and compiles it into executable JavaScript code. Using JavaScript's built-in *eval*-mechanism⁷, it then evaluates the JavaScript code to execute the query and to retrieve its result. Because JSINQ is a library and not a pre-processor, the compilation is done at runtime (or ideally at load-time).

2.2.1 Parsing query expressions

LINQ expressions are composed of query clauses which generally consist of keywords (such as *select*, *where*, *from*, etc.), identifiers and often arithmetic or logical expressions. In the following query expression, all three elements can be found:

⁷ The *eval*-function takes a string containing JavaScript code and tries to evaluate it. JSINQ does not actually use *eval* directly but instead uses the *Function* constructor which performs an *eval* internally.

```
from customer in customers
where customer.lastname.charAt(0) == 'S'
select customer.lastname
```

Listing 7: The components of a query expression

The words in bold printing are LINQ keywords, the underlined word is an identifier and the code that is not especially highlighted is made up of different types of expressions.

In order to compile a query expression, one has to break it down into query clauses first and the elements of each clause have to be separated. This could be done at least to some degree using regular expressions and in fact, an early version of JSINQ actually used regular expressions to parse the queries. However, this method turned out to be unreliable, because of the arbitrary JavaScript code that can stand between the query keywords. Since the JavaScript grammar is not regular, it is impossible to reliably separate query keywords from JavaScript code using only regular expressions and so the decision was made to drop this approach altogether in favor of a less trivial solution involving parser combinators.

2.2.2 Combinatory parsing

The basic idea behind combinatory parsing which has its roots in functional programming is that complex parsers can be constructed by means of combining simpler and ultimately primitive parsers. The word parser in this context is used to describe a function that receives a string and returns a parse tree along with any unconsumed input. Let us look at two simple parsers:

```
function a(input) {
  if (input.charAt(0) == 'a') {
    return ['a', input.substring(1)];
  } else {
    return null;
  }
}

function b(input) {
  if (input.charAt(0) == 'b') {
    return ['b', input.substring(1)];
  } else {
    return null;
  }
}
```

Listing 8: Two simple parsers

Both parsers accept a single character (“a” and “b”, respectively) and upon parsing said character return a tuple of the character itself and the remainder of the input string. The application of parser *a* to the input string “abba” would yield as its result a tuple of just the

character “a” and the remainder of the input which is “bba”. It is now not hard to imagine that more complex parsers can be constructed by repeatedly applying the parsers *a* and *b* to the input string and the input left unconsumed by the previous parser. However, there is an obvious asymmetry between the input a parser expects and the output it produces. To build a parser that accepts the input “abba” for example, we cannot write `a(b(b(a(“abba”)))` as this would pass the tuple returned by the innermost parser to the next parser in the chain of function calls which really expects a string as its input. To resolve this mismatch, we have to introduce another function that serves as an intermediary between two parsers.

```
function sequence(left, right) {
  return function(input) {
    var resultLeft = left(input);
    if (resultLeft == null) {
      return null;
    }
    var resultRight = right(resultLeft[1]);
    if (resultRight == null) {
      return null;
    }
    return [resultLeft[0] + resultRight[0], resultRight[1]];
  };
}
```

Listing 9: A sequential parser combinator

The function given Listing 9 takes two parsers and combines them sequentially. Using this parser combinator and the two parsers we defined earlier, we can now construct a new parser that recognizes the string “abba”:

```
var abba = sequence(sequence(sequence(a, b), b), a);
```

Listing 10: Combining parsers

The example above is identical to the EBNF production `abba = a b b a`. However, because our sequencing combinator uses prefix notation rather than infix notation, the equality of these two statements is not immediately apparent. JSINQ addresses this issue by selectively “overloading” the dot-operator to mean sequential combination⁸ as demonstrated in the following example.

```
this.abba = this.a().b().b().a();
```

Listing 11: The “abba” parser in JSINQ

⁸ This is not achieved using actual operator overloading (which JavaScript does not support) but by exploiting a multitude of JavaScript-specific language features.

Using only the sequencing combinator and the two parsers from before, we can already construct parsers for a theoretically infinite number of trivial grammars. However, we cannot yet express concepts like repetition, choice or optionality. Surprisingly, only a “handful” of additional parser combinators are needed to be able to construct parsers for any context-free grammar.

EBNF	JSINQ equivalent
<code>a = "a"</code>	<code>this.a = this.terminal("a");</code>
<code>ab = a b</code>	<code>this.ab = this.a().b();</code>
<code>c = a [b]</code>	<code>this.c = this.a().optional(this.b());</code>
<code>c = a {b}</code>	<code>this.c = this.a().zeroOrMore(this.b());</code>
<code>c = a {a}</code>	<code>this.c = this.oneOrMore(this.a())</code>
<code>d = a b c</code>	<code>this.d = this.oneOf(this.a(), this.b(), this.c());</code>
<code>d = ... d</code> (<i>* recursive parser *</i>)	<code>this.d = ...lazy(function() { return this.d(); })</code>

Table 1: JSINQ's set of parser combinators

The table above illustrates how EBNF productions can be translated into combinatory parsers. While the JavaScript code in the right-hand column reads a lot like the EBNF productions in the left-hand column, it is actually executable code in the sense that each row defines a fully functional parser. Using parser combinators, we can therefore construct complex parsers in a declarative fashion and without having to resort to using parser generators.

JSINQ uses combinatory parsers to recognize JavaScript expressions and to turn query expressions into parse trees. Creating these parsers (for the most part) is a matter of applying the translation rules described in Table 1 to portions of the grammar given in the C# Language Specification (Hejlsberg, 2007) as well as the ECMAScript Language Specification (Eich, 1999).

2.2.3 Compiling query expressions

A query is compiled into JavaScript code by means of repeatedly applying a set of transformation rules to its parse tree representation. The transformation process, which is described in the C# Language Specification (Hejlsberg, 2007), has much in common with peephole-optimization in compiler theory (Alfred V. Aho, 1986) where the generated code is inspected using a sliding window and particular sequences of machine code instructions are replaced with more efficient ones. Similarly, during query compilation the parse tree is traversed and particular sequences of query clauses are replaced either with other query clauses or with compiled JavaScript code.

The mapping from query expression syntax to method calls as described in 1.2 is not a direct one. While many query clauses have a direct counterpart in the set of “standard query

operators”, certain others do not. Most notably, query continuations are translated into nested *from*-clauses and multiple *from*-clauses are translated into calls to the *Select-Many*-method. Also, certain transformations may introduce “transparent identifiers”, compiler-generated variables that capture multiple range variables. These transparent identifiers have to follow certain rules regarding the scoping of their member variables and would therefore be difficult to support, if it was not for JavaScript’s infamous *with*-statement. The *with*-statement defines for a given object a block of code in which all of the object’s members can be accessed as if they were local variables, that is without explicitly referring to the object itself. The statement is sometimes considered harmful because of the potential ambiguity it may create with existing local variables. However, since the code generated by the JSINQ query compiler is not meant to be human-readable and because JSINQ only places the *with*-statement inside compiler-generated lambda expression, this ambiguity can easily be avoided.

3 Conclusion

In the age of rich internet applications, more and more business logic is moved from the server- into the client-tier. This effect is facilitated by the emergence of new and faster JavaScript runtimes. But while the client platform itself has reached a new maturity level and may very well be capable of hosting all this business logic, library support in many areas is still lacking. This is especially true for data processing which has previously been the exclusive domain of the server. JSINQ hopes to improve the developer experience in that area by providing a more declarative way of aggregating data.

Implementing LINQ to Objects in JavaScript has in many ways been an interesting experience. Most notably, the sheer expressiveness of the language with its lambda expressions, prototype-based inheritance and dynamic objects never ceases to amaze. But the implementation of JSINQ revealed to me as much about JavaScript as it did about LINQ and its roots in functional programming. Also, JSINQ marks my first use of combinatory parsers in “production-quality” code and for the most part, they do their job well.

References

Alfred V. Aho, R. S. (1986). *Compilers Principles, Techniques and Tools*. Addison-Wesley.

Chandra, J. (2007, July 16). *Javascript LINQ-like Querying*. Retrieved from Incoherent Rambling: http://netindonesia.net/blogs/jimmy/archive/2007/07/16/Javascript-LINQ_3F003F003F00_.aspx

Eich, B. (1999). *ECMAScript Language Specification 3rd Edition*. ECMA.

HBoss. (2008, October 2). *jLINQ*. Retrieved from jQuery Plugins: <http://plugins.jquery.com/project/jLINQ>

Hejlsberg, A. (2007). *C# Language Specification Version 3.0*. Microsoft Corporation.

Pietschmann, C. (2008, January 24). *LINQ to JavaScript*. Retrieved from Codeplex: <http://www.codeplex.com/JSLINQ>