


# Group E Midterm 2: JavaScript and Functional Programming



John Cox, Jesper Rage, Aahan  
Suneja, Dennis Honey, Yifang  
La, Jason Curt, Andrew Li,  
Pritpaul Padda, Jason Yu,  
Wesley Chow  
CSc 330  
Dr. Cheng

## TABLE OF CONTENTS

<b>SUMMARY</b>	<b>2</b>
<b>1. HISTORY</b>	<b>2</b>
1.1 JAVASCRIPT	2
1.2 LAMBDA CALCULUS	3
1.3 FUNCTIONAL PROGRAMMING	4
<b>2. LAZY EVALUATION IN JAVASCRIPT</b>	<b>5</b>
2.1 INTRODUCTION	5
2.2 IMPROVEMENTS	5
3.3 IMPLIMENTATIONS	6
<b>3. HIGHER-ORDER FUNCTIONS AND CLOSURE</b>	<b>8</b>
3.1 PURELY FUNCTIONAL LANGUAGES	8
3.2 FIRST-CLASS FUNCTIONS	
3.3 HIGHER-ORDER FUNCTION	#
3.4 JAVASCRIPT HIGER-ORDER FUNCTION SUPPORT	
3.5 CLOSURE	
3.6 JAVASCRIPT CLOSURE	
<b>4. LIST COMPREHENSION</b>	
4.1 LIST COMPREHENSION IN HASKELL	#
4.2 ARRAY COMPREHENSION IN JAVA	##
<b>CONCLUSION</b>	
<b>REFERENCES</b>	

## SUMMARY

The purpose of this report is to research and demonstrate our understanding of Lambda Calculus and Functional Programming as they relate to JavaScript. We will do this by covering four topics: History of JavaScript, lazy evaluation, higher-order functions and closure, and list comprehension.

This report was researched by ten group members over a seven hour open book exam. The information for the report was gathered using the internet and search engines.

## 1. HISTORY OF JAVASCRIPT

### JAVASCRIPT 1

JavaScript was developed by Brendan Eich and is an interpreted computer programming language as well as a prototype-based scripting language that uses dynamic typing and has first-class functions (the language supports passing functions as arguments to other functions) [1]. JavaScript was first developed under the name of Mocha, but was officially called LiveScript when it was shipped in beta releases for the Netscape Navigator 2.0 in September 1995, but was renamed into JavaScript when Netscape browser version 2.0B3 was released due to Netscape adding support for Java technology into its browsers. JavaScript's syntax is influenced by C, but it also takes many names and naming conventions from Java, despite the languages being unrelated to each other. Many of JavaScript's key design principles were influenced by the Self and Scheme programming languages and it's a multi-paradigm language (language that supports more than one programming paradigm) that supports object-oriented, imperative and functional programming styles[2].

#### JavaScript Version History:

Version	Release Date
1.0	March 1996
1.1	August 1996
1.2	June 1997
1.3	October 1998
1.4	
1.5	November 2000
1.6	November 2005
1.7	October 2006
1.8	June 2008
1.8.1	
1.8.2	June 2009
1.8.5	July 2010

---

#### LAMBDA CALCULUS 2

Lambda calculus is a formal system in mathematical logic and computer science that expresses computation based on function abstractions and application using variable binding and substitution. Lambda calculus was formulated by Alonzo Church in order to formalize the concept of effective computability i.e. Lambda calculus is an attempt to be precise about what computation actually is[3]. Lambda calculus was also created to investigate the Entscheidungsproblem, function definition, function application and recursion.

##### Formal Definition of Lambda Calculus:

Lambda expressions are composed of

- variables  $v_1, v_2, \dots, v_n, \dots$
- the abstraction symbols lambda ' $\lambda$ ' and dot '.'
- parentheses ( )

The set of lambda expressions,  $\Lambda$ , can be defined inductively:

1. If  $x$  is a variable, then  $x \in \Lambda$
2. If  $x$  is a variable and  $M \in \Lambda$ , then  $(\lambda x.M) \in \Lambda$
3. If  $M, N \in \Lambda$ , then  $(M N) \in \Lambda$

Instances of rule 2 are known as abstractions and instances of rule 3 are known as applications.

## Lambda Calculus and JavaScript:

Lambda calculus is at the core of many functional languages like Haskell, Scheme, Python, JavaScript and many more. The following diagram is the syntax in Lambda calculus and its equivalent in JavaScript:

	Lambda	JavaScript
variable	$x$	$x$
abstraction	$\lambda x.M$ or $\forall x.M$	<code>function(x){ return <math>M</math> }</code>
application	$((M) N)$	<code>(<math>M</math>)(<math>N</math>)</code>
declaration	$-$	<code>var a = <math>M</math>;</code>
syntactic sugar	$\lambda x \dots yz.M$ for $\lambda x \dots y. \lambda z.M$ $\lambda x.M N$ for $\lambda x.(M N)$ $(x M)$ for $((x) M)$ $(L M N)$ for $((L M) N)$	<code>function(x,...,y,z){ return <math>M</math> }</code> for <code>function(x,...,y){ return function(z){ return <math>M</math> } }</code> <code>x(<math>M</math>)</code> for <code>(x)(<math>M</math>)</code> <code>L(<math>M</math>)(<math>N</math>)</code> for <code>(L(<math>M</math>))(<math>N</math>)</code>

## FUNCTIONAL PROGRAMMING 3

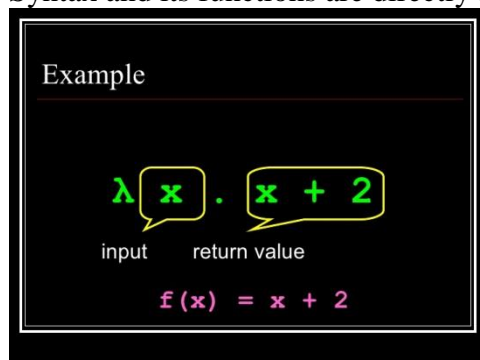
Functional programming is a programming paradigm, a style of building the structure and elements of computer programs that treats computation as the evaluation of mathematical functions and avoids state and mutable data[4]. Functional programming emphasizes that functions produce the results depending on the input and not the program state i.e. these functions are purely mathematical. Functional programming is also a declarative programming paradigm, which means that programming is done with expressions.

JavaScript connection to functional programming:

In JavaScript, functions could be assigned to variables just like any other data types.

Functions can accept another function as a parameter and also return a function.

In addition, it can also allow functions with no names. JavaScript and functional programming are interconnected because it uses a Lambda Syntax and its functions are directly available as values[9].



JavaScript is both a scripting and a functional language as it contains the main characteristics of the Functional Paradigm. The main characteristics are: Contains support for anonymous **Lambda Calculus** functions, Complex algorithms can be implemented from simple algorithms (**Compositionality**), **Formula** (calculations), **Recursion** using higher-order functions, **Referential Transparency** (the return value of a function depends solely on its arguments, if at all) and JavaScript has no **Side Effects**[10].

## 2. LAZY EVALUATION IN JAVASCRIPT

### INTRODUCTION 1

Lazy evaluation is a strategy where evaluations of expressions are delayed by default and are not computed unless otherwise needed. Introduced by Christopher Wadsworth in 1971[12], this would be used to evaluate lambda calculus expressions by evaluating the lambda functions before evaluating the arguments. Lazy evaluation would be used in functional programming languages today to improve runtime speeds. It does not even consider unnecessary conditions such as the second condition in a compound check where the first condition required already evaluates false, the second condition is not evaluated, saving resources and time[3].

### IMPROVEMENTS 2

Call-by-value (or eager evaluation) would evaluate and simplify all expressions before evaluating the function. Lazy evaluation on the other hand would only evaluate functions by default and only evaluate expressions needed to continue with the computing. This is relevant to JavaScript where traversing the web requires evaluation of all possible destinations per page[15]. Exponential reductions in running times are possible with lazy evaluation in web specific scenarios. For example, a website containing many images will take relatively longer to load than the Google homepage if a page does not need to load all the images all at once.

### IMPLIMENTATIONS 3

There are three libraries that are implemented in javascript that use lazy evaluation.

#### 1. stream.js

This is an add-on to the standard javascript library which adds a data structure that implements lists of infinite length\*. Example code that demonstrates the power of this library is below:

Code snippet #1

```
function ones() {  
    return new Stream(1,ones);  
}  
var n = 100;  
var s = ones();  
s.take(n).print();
```

The function *ones()* will create a stream that contains an infinite list of 1's (ie. [1, 1, 1, 1, ...]) that can be made on demand. The *take(n).print()* function will grab the first *n* elements of the stream *s*, then print those *n* elements taken from the stream. As evaluation of expressions are only done by demand, space complexity will significantly decrease for larger values of *n*. Compare it to the following code written imperatively:

Code snippet #2

```
function ones (a) {  
    var list = new Array();  
    for (var i = 0; i < a; i++) {  
        list[i] = 1;  
    }  
    return list;  
}  
var n = 100  
var s = ones(n);  
s.toString();
```

As the code is not lazily evaluated, it will create an array of size 100 and fill it with 1s before printing it; on the other hand, the idea of an infinite list is defined in the function *ones()* and will create the list “on-demand” as *s.take(n).print()* creates it and prints it. If we use a relative timeline to show the space allocation between code snippet #1 and code snippet #2, the space usage between the two snippets are approximately the same at the end of the programs:

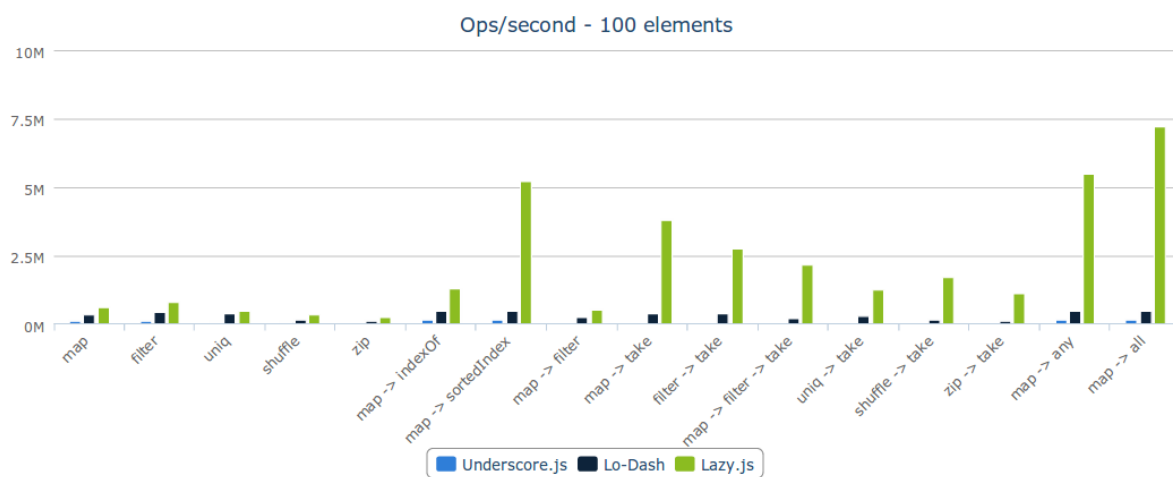
#### TIMELINE OF BOTH PROGRAMS



During runtime, it is clear that memory usage by the second code snippet was larger than the first code snippet. This is one advantage that lazy evaluation has over non-lazy evaluation: no allocation to memory immediately for a data structure.

## 2. lazy.js

lazy.js is a utility library that offers tools for developing web applications. This is similar to Prototype.js and Underscore.js[13] but it is implemented with lazy evaluation. The speed differences already differ with 100 operations, by determined by this chart:



The functions as a stand-alone do not produce amazing results in terms of operations but compositions of functions show a vast difference in operations computed per second.

## 3. linq.js

This implements all the .NET 4.0 methods[14] and LINQ with lazy evaluation. LINQ (Language-Integrated Query) is a library in Visual Studio 2008 that extends functionality to C# and Visual Basic[16]. It is used for updating and storing data in databases. As linq.js has a policy of lazy evaluation, updating and storing data would be faster as any data that does not necessarily need to be stored or updated.

## 3. HIGHER-ORDER FUNCTIONS AND CLOSURE

### PURELY FUNCTIONAL LANGUAGES 1

A pure function is one which has no side effects — it takes a value in and gives a value back. There's no global state that functions modify. Purity has a number of interesting consequences, such as the fact that



evaluation can be lazy — since a function call has no purpose but to return a value, then you don't need to actually execute the function if you aren't going to use its value.

Another consequence is that it doesn't matter in what order functions are evaluated — since they can't affect each other, you can do them in any order that's convenient. This means that some of the problems posed by parallel programming simply don't exist, since there isn't a "wrong" or "right" order for functions to execute.

A functional programming language is one in which functions are first-class. In other words, you can manipulate functions with exactly the same ease with which you can manipulate all other first-class values[19].

---

## FIRST-CLASS FUNCTIONS 2

The term “first-class” means that something is just a value. A first-class function is one that can go anywhere that any other value can go—there are few to no restrictions. A number in JavaScript is surely a first-class thing, and therefore a first-class function has a similar nature:

- A function can be stored in a variable.
- A function can be stored in an array slot.
- A function can be stored in an object field.
- A function can be created as needed, anywhere in the program.
- **A function can be passed on to another function.**
- **A function can be returned from another function.**

The last two points define by example what we could call a “higher-order” function [20].

---

## HIGHER-ORDER FUNCTIONS 3

As mentioned above, a higher-order function is a first-class function which takes a function as an argument and returns a function as a result.

Higher-order functions can be used to generalise many algorithms that regular functions cannot easily describe. When you have a repertoire of these functions at your disposal, it can help you think about your code in a clearer way: Instead of a messy set of variables and loops, you can decompose algorithms into a combination of a few fundamental algorithms, which are invoked by name, and do not have to be typed out again and again[22].

---

## JAVASCRIPT HIGHER-ORDER FUNCTION SUPPORT 4

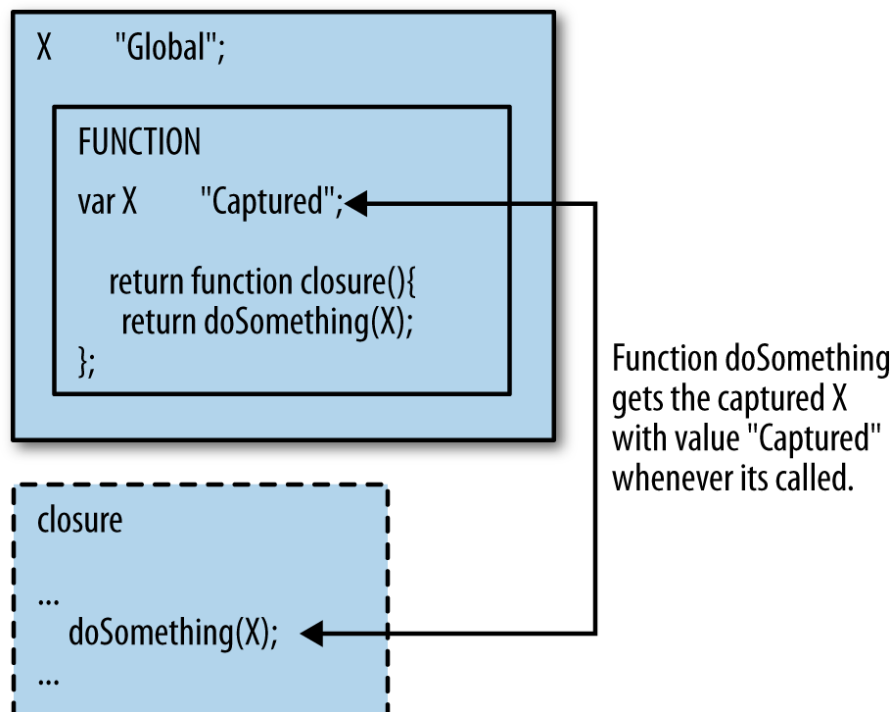
JavaScript implemented map, reduce and filter functions in their Array class in JavaScript 1.6. Since there is no lazy evaluation in

JavaScript, these can't be used on infinite lists. They can be used only on finite arrays[21].

## CLOSURE 5

A variable that is assigned to a function is a *structure* in memory that is used to represent the function. For many languages, this *structure* is the memory address of the function, resulting in the variable only being able to invoke the function. JavaScript implements a slightly more complex setup for the *structure*. In JavaScript, the *structure* holds both the address of the function and its *referencing environment*. The referencing environment means the environment the function was originally declared in. If the function was originally declared within another function, then the referencing environment will mean the function that the original function was declared in.

Having access to the referencing environment means access to certain non-local variables. Normally, a function has access to its own local variables and to the global variables. With the *referencing environment* at the structure's disposal, calls to certain non-local variables (i.e. variables that were not declared in the function, but were declared in the function where it was defined) are possible. JavaScript implementers refer to the *referencing environment* as the function's *context*. When the *structure* that points to a function also contains its original context (*referencing environment*), it is called a closure.



---

## JAVASCRIPT CLOSURE 6

First, we have to cover an important topic: scope.

A variable's scope defines where the variable can be used. In JavaScript there is really only local scope and global scope. Notice that local variables will hide global variables of the same name.

Nesting a function inside a function is essentially nesting a local scope inside a local scope. Note that functions have access to all the variables and arguments in their ancestor's scope.

```
var x = 10;  
function f(){  
    var y = 15;  
    function g(){  
        var z = 25;  
        alert(x+y+z);  
    }  
    g();  
}  
f(); // =>50
```

The above behavior is what we expect after reading section 10 of the ECMAScript standard. JavaScript creates a new activation object for each scope. The activation object references all function arguments and variables in the scope. The activation objects become part of a scope chain that JavaScript will traverse when trying to find a variable.

When we ask for the value of *x* inside function *g()*, JavaScript looks at activation object for the innermost scope, but doesn't find *x*, so it moves to the parent scope, but still doesn't find *x*. Finally, the runtime looks at the global scope and finds *x*. If the engine had found *x* in a nested scope, it wouldn't have gotten to the *x* in the global scope.

```
function sayHello(name) {  
    var text = 'Hello ' + name; // local variable  
    var sayAlert = function() { alert(text); }  
    return sayAlert;  
}  
  
var say = sayHello('Dr. Cheng');  
say();//=> Hello Dr.Cheng
```

Returning a function object creates a closure. Generally speaking, when a function like `sayHello()` exits, all the local variables and arguments are lost. A closure, however, closes over its environment and keeps these local variables and arguments alive. The function captures its execution environment.

It's easy to see how a JavaScript engine might implement a closure. All a nested function has to do is carry a reference to its activation object, which will keep all the variables and arguments in the scope chain alive[20].

## 4. LIST COMPREHENSION

### LIST COMPREHENSION IN HASKELL 1

List comprehensions in Haskell are considered syntactic sugar. Meaning, although they are used for special notations for special applications, they do not add any extra functionality to the language. They instead provide textual replacements that provide simplicity to the language and maximize readability. For example `[1,2,3,4,5]` is sugar for `(1:2:3:4:5:[])` and `do x <- f; g x` is sugar for `f >>= (\x -> g x)`. Although syntactic sugar is generally considered a positive practice, too much of it can cause underlying semantics to become unclear. However, Haskell achieves a balance of syntactic sugar where the language itself is quite flexible but still achieves clarity and conciseness[18].

In Haskell list comprehension is a syntactic construct that uses the form of mathematical set-builder notation for creating lists based on existing lists. The following is best explained in an example:

$$S = \{2 \bullet x \mid x \in \mathbb{N}, x^2 > 3\}$$

This expression in set-builder notation states that the set  $S$  contains the set of all numbers 2 times  $x$ , where  $x$  is a member of natural numbers and  $x^2$  is greater than 3.  $2 \bullet x$  refers to the output function,  $x$  is the variable,  $\mathbb{N}$  is the input set, and  $x^2 > 3$  is the predicate. We can represent this expression in Haskell using very similar syntax,

$$S = [2 * x \mid x <- [0..], x^2 > 3]$$

In Haskell,  $2 * x$  refers to the output function,  $x$  is the variable,  $[0..]$  represents all natural numbers, which is our input set, and  $x^2 > 3$  represents the predicate. List comprehension in Haskell requires the support of generators and predicates. In our example, the phrase  $x <- [0..]$  is a generator that states that all  $x$  are drawn from the set of natural numbers. In our case we only have one generator, but more than one is allowed, as in: `[(x,y) | x <- x1, y <- y1]`. Suppose  $x1$  is `[1,2]` and  $y1$  is `[3,4]`, then the result is `[(1,3),(1,4),(2,3),(2,4)]`[17].

Haskell's list comprehension is intuitive and akin to generating functions in algebraic math. For further examples, suppose we have the following functions:

$$t = [x^2 \mid x <- [0..], x \% 2 == 0]$$

Here we have  $t$  as the resulting list,  $x^2$  as the output function,  $x <- [0..]$  as the input set, which denotes  $x$  is the set of all natural numbers, and  $x \% 2 == 0$  as the predicate, which filters all  $x$  values. In this case it limits  $x$  values to even numbers. Thus  $t$  gives the list of all the squares of the natural numbers that are even,  $t = [0, 4, 16, 36, 64, 100, 144, 196, 256, \dots]$  (infinite sequence)]. Multiple generators are possible in Haskell, separated by commas. For example, if we have:

$[(x,y) \mid x <- [0,2], y <- [0,2]]$

Then we will obtain  $[(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)]$ , which is analogous to the construction of ordered pairs in math (and extends to  $n$ -tuples, should we have more generators)[17].

---

## ARRAY COMPREHENSION IN JAVA 2

### JavaScript supports Generators

Pythonic generators along with iterators were first introduced in the 1.7 version of JavaScript. Generators are special functions that create iterator objects. Generators as functions can take arguments, which are bounded the first time the function is called. Also, they are not always executed when they are called, instead they return iterator objects, which are objects that are able to access and keep track of items in a sequence. Generators enable lazy computation of sequences, with items calculated on demand as they are needed. Generator expressions are syntactically almost identical to array comprehensions - they use parentheses instead of braces - but instead of building an array they create a generator that can execute lazily[22].

### JavaScript Array Comprehension

Array comprehensions were implemented in JavaScript 1.7. They are a well-understood and popular language feature of list comprehensions, found in languages like python and Haskell. They are also inspired by mathematical notations of set comprehensions. Array comprehensions are also convenient, declarative form of creating computed arrays with a literal syntax that reads naturally. A drawback of array comprehension is that they create an entire new array to be constructed in memory. This is fine when the input is small, the overhead involved is insignificant, but when the input is a large, the creation of a new array can be problematic[22].

### Examples

Here are a few examples of array comprehension in use:

- Filtering an array:  $[x \text{ for } (x \text{ of } a) \text{ if } (x.\text{color} === \text{'blue'})]$
- Mapping an array:  $[\text{square}(x) \text{ for } (x \text{ of } [1,2,3,4,5])]$
- Cartesian product:  $[[i,j] \text{ for } (i \text{ of rows}) \text{ for } (j \text{ of columns})]$

Comprehensions can also be used to select items that match a particular expression. Here is a comprehension which selects only odd numbers.

```
var numbers = [1,2,3,21,22,30];
```

```
var odds = [i for (i of numbers) if (i % 2 === 1)];  
alert(odds); // alerts 1, 3, 21
```

The square brackets of an array comprehension introduce an implicit block for scoping purposes. New variables are treated as if they had been declared. This means that they will not be available outside of the expression they were used in. The input to an array comprehension expression does not need to be an array; iterators and generators can also be used.

---

### HASKELL AND JAVASCRIPT 3

Haskell and JavaScript's list/array comprehension essentially works the same way, but the code each language uses is quite different. Haskell's list generation's format is very much like generating functions in math and thus is more intuitive and is generally easier for users with a background in mathematics to understand[22]. Such as the math expression in set-builder notation of  $S = \{2 \bullet x \mid x \in \mathbb{N}, x^2 > 3\}$  is similar to the Haskell version of the same set:  $S = [2 * x \mid x \leftarrow [0..], x^2 > 3]$ . JavaScript, on the other hand, opts for a more computing approach, using the keywords "for", "each", "if", "" etc. For example, these two lines of code shown below double the values of the integers in the array:

```
var numbers = [1, 2, 3, 4];
```

```
var doubled = [i * 2 for (i of numbers)];
```

For direct comparison consider the examples that return even numbers from zero to ten:

Haskell:

```
even = [n | n <- [0..10], n `mod` 2 == 0]
```

JavaScript:

```
var numbers = [0,1,2,3,4,5,6,7,8,9,10];
```

```
var even = [i for (i of numbers) if (i % 2 === 0)];
```

As you can see, Haskell provides apt readability and concise syntax, but sometimes the shortened code can cause confusion[22]. On the other hand, JavaScript's syntax is a bit more complicated as it doesn't allow short hand array declarations, but it's more readable.

In direct comparison, JavaScript is more expressive for reasons mentioned above, but it requires you to understand the technical terms and the technicalities that come with each keyword. Whereas Haskell is much more concise but also can be confusing with its minimal syntax.

## REFERENCES

- [1] Wikipedia, "First Class Function," [Online]. Available: [http://en.wikipedia.org/wiki/First-class\\_functions](http://en.wikipedia.org/wiki/First-class_functions). [Accessed 30 10 2013].
- [2] Wikipedia, "JavaScript," [Online]. Available: <http://en.wikipedia.org/wiki/JavaScript>. [Accessed 30 10 2013].
- [3] M. James, "Lambda Calculus for Programmers," [Online]. Available: <http://www.i-programmer.info/programming/theory/4514-lambda-calculus-for-programmers.html>. [Accessed 30 10 2013].
- [4] Wikipedia, "Functional Programming," [Online]. Available: [http://en.wikipedia.org/wiki/Functional\\_programming](http://en.wikipedia.org/wiki/Functional_programming). [Accessed 30 10 2013].
- [5] "Lambda Calculus in JavaScript Syntax," [Online]. Available: <http://labs.orezdu.org/lambda/>. [Accessed 30 10 2013].
- [6] Wikipedia, "Lambda Calculus," [Online]. Available: [http://en.wikipedia.org/wiki/Lambda\\_calculus](http://en.wikipedia.org/wiki/Lambda_calculus). [Accessed 30 10 2013].
- [7] T. Montgomery, "Functional Programming," [Online]. Available: <http://www.slideshare.net/tmont/introduction-to-functional-programming-in-javascript>. [Accessed 30 10 2013].
- [8] S. Rajan, "Functional JavaScript," [Online]. Available: <http://functionaljavascript.blogspot.ca/2013/03/introduction-to-functional-javascript.html>. [Accessed 30 10 2013].
- [9] A. Kamat, "4 Powerful Features of JavaScript Programming Language," [Online]. Available: <http://www.webdesign.org/web-programming/javascript/4-powerful-features-of-javascript-programming-language.17008.html#ixzz2jGqQPnDE>. [Accessed 30 10 2013].
- [10] Wikipedia, "Comparison of Programming Languages," [Online]. Available: [http://en.wikipedia.org/wiki/Comparison\\_of\\_programming\\_paradigms](http://en.wikipedia.org/wiki/Comparison_of_programming_paradigms). [Accessed 30 10 2013].
- [11] Unknown, "Stream.js," [Online]. Available: <http://streamjs.org/>. [Accessed 30 10 2013].

- [12] Wikipedia, "Christopher P. Wadsworth," [Online]. Available: [http://en.wikipedia.org/wiki/Christopher\\_P.\\_Wadsworth](http://en.wikipedia.org/wiki/Christopher_P._Wadsworth). [Accessed 30 10 2013].
- [13] D. Tao, "Like Underscore, but Lazier," [Online]. Available: <http://danieltao.com/lazy.js/>. [Accessed 30 10 2013].
- [14] CodePlex, "linq.js - LINQ for JavaScript," [Online]. Available: <http://linqjs.codeplex.com/>. [Accessed 30 10 2013].
- [15] Stoyan, "Lazy HTML Evaluation," [Online]. Available: <http://www.phpied.com/lazy-html-evaluation/>. [Accessed 30 10 2013].
- [16] Microsoft, "LINQ (Language-Integrated Query)," [Online]. Available: <http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx>. [Accessed 30 10 2013].
- [17] Wikipedia, "List Comprehension," [Online]. Available: [http://en.wikipedia.org/wiki/List\\_comprehension#Haskell](http://en.wikipedia.org/wiki/List_comprehension#Haskell). [Accessed 30 10 2013].
- [18] C. Gibbard, "Syntactic Sugar," [Online]. Available: [http://www.haskell.org/haskellwiki/Syntactic\\_sugar](http://www.haskell.org/haskellwiki/Syntactic_sugar). [Accessed 30 10 2013].
- [19] B. S, "Pure Functional Language: Haskell," [Online]. Available: <http://stackoverflow.com/questions/4382223/pure-functional-language-haskell>. [Accessed 30 10 2013].
- [20] M. Hogue, "Closure," in Functional JavaScript.
- [21] M. Haverbeke, "Higher-Order Functions," in Eloquent JavaScript: A Modern Introduction to Programming.
- [22] M. Foundation, "JavaScript Guide," Mozilla Developer Network, [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>. [Accessed 30 10 2013].