University of Victoria

JavaScript and Functional Programming

Prepared by Team A for CSC 330

JB Braendel
Jessica Paquette
John Rogers
Marvin Messing
Ryan McDonald
Gabriel Letourneau
Catlin Zilinski
Jessica Blue
Hiroki Yun
Bill Xiong

Table of Contents

History

Origins

It was a beautiful spring morning, in April of 1995. In Mountain View, California, Netscape had hired Brendan Eich to build an interpreted language to be run on the browser. Netscape was at war with Microsoft over the future of the web, which was then in its infancy [1]. This was the age of the browser wars; Netscape's Mosaic (later renamed to Netscape Navigator) held the highest distribution of users, which was over 80% [20]. Microsoft's Internet Explorer 2.0 was then released for free, and available to all Windows users. Netscape's release of JavaScript was seen as a spark of genius, and was quickly reverse engineered by Microsoft to create their clone language, JScript [5]. Since C++ and Java were already aimed at the professional sector, Netscape wanted a lightweight language that complemented Java and appealed to the nonprofessional programmer, in order to increase market share [21]. Brendan started with a vision: to "bring Scheme to the browser." He would call this language Mocha. The name would be changed to LiveScript only six months later, and then to JavaScript shortly after Sun Microsystems and Netscape started a license agreement in early December [22]. Since Java was the hip programming language at the time, management had stated that this language must look and feel like Java. Eich had grudgingly thrown together a Java-like language on the outside, but with a Scheme and Self-like structure on the inside; hence, JavaScript retained some of the prototype-based, object-oriented style of Self and some of the functional qualities of Scheme [16]. Because the language had been built in only 10 days, bugs were inevitable; Eich blames a number of JavaScript's idiosyncrasies on this short deadline [11][16]. Those bugs were, in turn, carelessly added into Microsoft's JScript during its development, and Microsoft has insisted on keeping those mistakes as a part of the ECMAScript standard to avoid breaking currently running programs that depended on those bugs [13].

JavaScript was first released with Netscape 2.0B3. While growth was initially steady, JavaScript added unparalleled functionality to the boring old browser at the time. This allowed Netscape to pull ahead in the browser wars, eventually leading to the obligatory Microsoft-developed JScript. The battle had taken a turn when Netscape announced it would submit JavaScript to ECMA International for standardization across the web; and so began the ECMA 262 (ECMAScript) committee to standardize the language across all browsers and interpreters. The development of ECMAScript began in November of 1996, and was first adopted June 1997 [11].

While originally shunned by professional programmers due to its large appeal to newer programmers, JavaScript was the first lambda language to go mainstream. Today, it is apparent that Netscape had won the war: JavaScript is used on 89.3% of all websites [4], and supported by default on all major browsers. It is a large part of what makes up the client-side user experience, through both the traditional computer access channel and newer channels such as the tablet and the smartphone [16]. Variants such as node.js have made splashes server-side, and are also growing in popularity [3]. It is wildly popular at Google and other web service giants, and forms the backbone of many widely used web applications [18].

Functional Characteristics

JavaScript is heavily influenced by and was originally very syntactically similar to the languages Scheme and Self. Netscape ordered the syntax to be redone later in development to make it more similar to Java. The Scheme-like language dynamics were however left unchanged [12].

Because JavaScript was heavily influenced by Scheme, it shares many similarities and maintains many of the functional characteristics of Scheme. Both languages have First Class Functions, meaning functions can be passed, returned, and stored just like any other value [8]. Closures were also adopted from Scheme, so a function defined inside another function can access local variables defined in the outer function [16]. JavaScript is also lexically scoped as opposed to dynamically scoped [9]. This means the interpreter searches in textually surrounding definitions. Lastly, Scheme and JavaScript also allow global variables to be implicitly defined at the global level [12]. jQuery is an example of a well-known and widely used library that effectively utilizes JavaScript's functional programming features [23]. The root jQuery object that you call in the library can be qualified as a monad, a structure that represents computations defined as sequences of steps [24], and a monad is a fundamental concept of functional programming languages.

Expressiveness

The expressiveness of a language can be thought of in terms of either the user or the compiler. In terms of the user, an expressive language is one that is readable, compact, powerful, and succinct. A powerful language is one that can solve a number of different problems. In terms of the compiler, expressiveness is best described at how well it can accurately interpret the code. Since JavaScript was written with the non-programmer oriented web designers in mind, it was built with a very simple and readable syntax. Popular library jQuery makes JavaScript more expressive by providing functions which solve many common problems (eg. $.animate, $.addClass and $.removeClass, $.clone, $.ajax, etc.). This helps designers in reading, writing and manipulating JavaScript code for their websites by making the code more compact and readable. With the recent release of the HTML5 specification, JavaScript is becoming more powerful by being able to work with a number of new features which are now being adopted into the web: The Canvas, WebGL, Media Queries, FileBlobs, WebRTC, Geolocation, and IndexDB are all examples of new features to the JavaScript API which allow developers to solve more problems and extend their web applications further. JavaScript can also run on the server side (e.g. node.js) to solve server-related problems too.

Lazy Evaluation

JavaScript does not natively support lazy evaluation, but simulating it is becoming increasingly common amongst JavaScript developers. The prospects of infinite data structures, absence of unnecessary calculations, and easily readable code has resulted in several libraries which simulate laziness. We will outline libraries which implement, or use, such libraries and their impact on the code produced in terms of conciseness, expressiveness, and maintainability. Afterwards, we will explore the feasibility of implementing a fully lazy dialect of JavaScript.

Lazy.js

Before discussing the benefits of Lazy Evaluation in JavaScript, we will introduce a sample of how laziness is currently being simulated in JavaScript. *Lazy.js* is a utility library which simulates laziness in JavaScript by delaying the evaluation of functions. Such a delay may be implemented by using the code shown below.

```
function(){return my_delayed_function()}
```

As a result of delayed evaluation, array comprehension is an easy and efficient task. The desired mapping and filtering functions are passed to the Lazy module, instead of being applied inside of a messy for loop in the middle of the code, as is shown in the figure below. The code on the left provides the efficiency of the code on the right by making use of sequences, and will not generate more than one array to calculate the correct result.

With Lazy.js	Without Lazy.js

`var result = Lazy(array)`
`  .map(myfunction)`
`  .filter(myfilter)`
`  .take(somenumber);`
	`var results = [];`
	`for (var i = 0; i < array.length; ++i) {`
	`  var value = myfunction(array[i]);`
	`  if (!myfilter(value)) {`
	`  results.push(value);`
	`  if (results.length === somenumber) {`
	`  break;`
	`  }`
	`  }`
	`}`

[34]

The Lazy module can also be used to generate indefinite sequences, such as random numbers between one and one-thousand. This is also useful when parsing strings, as the string is treated as a sequence, and the key function is not applied to the entire string, but only the desired portion of the string.

```
var unlimitedRandomNumbers = Lazy.generate(function() { return Math.random(); })
  .map(function(e) { return Math.floor(e * 1000) + 1; })
  .take(somenumber);

var firstFiveLines = Lazy(text).split("\n").take(5);
```

[34]

Lazy Evaluation and node.js

Lazy evaluation concepts have been shown to be useful in expanding the expressive capabilities of JavaScript with the rising popularity of node.js. Node.js is a JavaScript platform for Chrome whose goals include building “fast, scalable network applications”. [37] Node.js is designed with event-driven non-blocking I/O, simplicity, and efficiency in mind, while requiring that it handle distributed data-intensive real-time applications.

JavaScript has no native support for threads, which is not a hindrance to its use as a networking language as, “thread-based networking is relatively inefficient and very difficult to use.” [37] Instead of using threads node.js takes an event-based approach, where it enters an event loop without performing a blocking call, and remains in that loop until no further events exist to be handled.

The event model used by node.js applications is well-handled by functional programming paradigms, as the number of events to be handled is undefined. Consider returning a list from an asynchronous function without lazy evaluation. The data cannot be assumed to exist in the list until every event that is to write to the structure has finished. Therefore, every time the list return event is accessed the program must check for the completion of every write event. The model provided by traditional JavaScript works, but we lose the ability to formulate pipes and chain functions. The result is code whose purpose is obscured, and which does not properly express the given problem.

If lazy evaluation is supported, then the events do not have to all complete before returning the list. Instead, it is possible to return the elements of the list as events complete, on an as-needed basis. This removes the necessity to constantly check the status of each event. We can simply describe the list as what it is: some list of some length, which will be populated with data provided by some number of events. The purely functional code is a far better description of the problem, more understandable, and more maintainable. [38]

JavaScript programmers have acknowledged this, and have created several libraries and engines to simulate lazy evaluation, among other concepts borrowed from popular functional programming languages. A common approach to handling the previous solution is by introducing the concept of *streams*. A stream is simply a list which may have infinite elements, and is always evaluated lazily. Streams have been implemented for use in JavaScript in Node-Lazy, stream.js among others. The nature of streams allows for the creation of code which handles large asynchronous structures elegantly and expressively.

Below is a comparison between two code snippets. The left-hand side consists of code written using the Node-Lazy library, and the right-hand side consists of code written without the use of any lazy evaluation.

With Node-Lazy	Without

`var Store = require('supermarket');`
 	
`var db = new Store({ filename : 'users.db', json : true });`
 	
`db.filter(function (user, meta) {`
`  return meta.age > 20;`
`})`
`.take(5)`
`.join(function (xs) {`
`// xs contains the first 5 users who are over 20!`
`});`
	`var Store = require('supermarket');`
	
	`var db = new Store({ filename : 'users.db', json : true });`
	
	`var users_over_20 = [];`
	`db.filter(`
	`function (user, meta) {`
	`// predicate function`
	`return meta.age > 20;`
	`},`
	`function (err, user, meta) {`
	`// function that gets executed when predicate is true`
	`if (users_over_20.length < 5)`
	`users_over_20.push(meta);`
	`},`
	`function () {`
	`// done function, called when all records have been filtered`
	
	`// now do something with users_over_20`
	`}`
	`)`

Selects five users from a database who are over 20 years old and stores them in users_over_20 [38]

The code featuring lazy-evaluation is just over half as long as the standard code. It is self-documenting, easy to read, and highly maintainable. Conversely, the code without lazy evaluation is heavily reliant on function calls and error checking, and is far more difficult to describe than the lazy code.

Implementing Fully Lazy Languages

Implementing a fully lazy JavaScript would be significantly more complicated than the traditional method of interpretation. Currently, there have been no attempts at such an implementation.

Fully lazy languages cannot be expressed by a normal finite state automata. Instead, a new structure must be implemented known as a *fully lazy functional machine* (FLFM). This machine

can be directly translated to bytecode for interpretation. Furthermore, a fully lazy language may be translated into a FLFM.

The given functional language must be translated to a *fully lazy normal form* (FLNF). This normal form must include

- Syntactic domains for basic values, identifiers, and expressions
- The abstract syntax of the language
- Semantic domains for basic values, expressible values, pairs, functions, denotable values, and environments
- The syntax of the fully lazy normal form
- A context condition for the fully lazy normal form

Additionally, the FLFM must maintain a stack of values, where

- the first value is evaluated on the top of the stack, and then applied to the remaining values
- values may be *closures* or constants

We define a closure as a pair which contains the code for evaluating a list of expressions (*e0, e1, e2,..., en*), and the current environment. Whenever we remove values from the stack, if there are any left over, we extend the current environment using those values such that each value *ai* is bound to *xi* in an expression fn *x1…xk e*. Evaluation occurs whenever we find a sufficient, set number of values on the stack.

Values are categorized as *evaluated* or *unevaluated.* Evaluated values are never evaluated again, and unevaluated values may be evaluated once, and then are marked as such. In the case of a partial evaluation, we store the resulting function as a closure.

The FLFM has four registers: S for the stack, E for the environment, C for the control code, and D for the dump. Closures are implemented as consisting of a code *c,* an environment *e,* and an empty list. The FLFM may perform load operations, alter environments, evaluate, apply, call a new code, and update. Also supported are primitive arithmetic and boolean operations, and list operations. Conditional operations are expressed as combinations.

An FLFM may be implemented on a physical machine with the use of the system stack, a garbage collector, and two heaps. The current code is controlled by the program counter. The dump is embedded into the system stack using the current stack frame in order to perform recursive calls to functions. The value that would normally be on the top of the stack conceptually is held in a register instead. Since environments are shared, they are implemented as a list. Each value consists of a tag denoting its type, and either a primitive value or a pointer to some cell in the heap storage. Only one of the two heaps is used for storage at a time. When the active storage heap is full, the values in the two heaps are switched, and the other heap becomes the active storage heap. An FLFM implemented in this way on a Motorola 68000 was

able to expand most of its instructions directly into valid 68000 instructions during interpretation.[41]

Higher-order Functions and Closure

What is a closure?

In relation to Lambda calculus, closure is a method of binding free variables to ensure they are separate from the calling scope [25]. For example, in Figure 1 we create a closure to bind the otherwise free variable y.

```
λx.xy => λy.(λx.xy)
```

Figure 1: example of Lambda expression representing closure.

In computer science, a closure is defined to include two things: a function and an environment, which consists of any local variables that were in scope at the time that the closure was created [26]. These variables can be referenced by the function even after their scope has terminated. Closures were developed in the 1960s and first implemented as a feature of Scheme as a method for supporting lexically scoped first-class functions [27] (see Higher-order Functions section for definition). It's worth mentioning that closures go hand in hand with first-class functions. Languages without first-class functions can still support closures (see Figure 4), but they are often greatly stunted. Closures, paired with first-class functions, allow programmers to pass around and store functions, while still maintaining the lexical scope in which the function was defined. Thankfully, JavaScript supports first-class functions so its closures are a powerful way to pass around ad-hoc encapsulated states and are used extensively in implementing callbacks for user driven events.

How does JavaScript implement closures?

JavaScript code is most often user-event driven by the use of events and attaching listeners to execute some action on a specified event. These event listeners are attached to events as closures, often by creating anonymous functions. In fact, a report that looked at 10 large JavaScript projects found that only 7% of functions in the code were named [29]. As a result, closures are used extensively throughout JavaScript and can be created in one of two ways [26]:

1. By returning a function that was defined within a called function
2. By creating an anonymous function and assigning it to a variable

Both methods achieve the same results, they both create an object that acts like a function but contains a copy the lexical scope in which the function being closed was defined [28]. When the closure is called, JavaScript uses the scope bound to the closure to resolve non-local variables referenced within the function. The way in which JavaScript implements its closures, binding a

new copy of the lexical scope each time a closure is created, provides for the ability to actually modify the scope bound to the closure as shown in the following example.

Code:

```
function createClosure(value, name) {
    var name = name+" Closure!";
    function closure() {
        value++;  // Increments the value in the bound scope
        console.log(name+" : "+value);
    };
    return closure;
}
var x = createClosure(5, "x");
x();
x();
y = createClosure(2, "y");
y();
y();
x = y;
x();
x();
```

Output:

```
x Closure! : 6
x Closure! : 7
y Closure! : 3
y Closure! : 4
y Closure! : 5
y Closure! : 6
```

Figure 1. Example of creating a closure, and the binding of scope with JavaScript.

There are, however, two problems with JavaScript's implementation of closures, which violate the constraints present in the concept of a closure in lambda calculus.

The first problem with JavaScript's implementation of closures is in its handling of global variables referenced from within the closure. By definition, a closure is supposed to be self-contained, it should be free of any dependency on the environment in which the closure is executed. If a closure in JavaScript references a global variable, the self-contained aspect of the closure breaks down. In JavaScript, the global scope is not stored as part of the scope bound to the closure, so when the global variable is referenced the value depends on the current global scope. This global reference behaviour violates how a closure is supposed to behave, and is shown in the following behaviour.

Code:

```
var glob = "Global";
function createClosure(name) {
    function closure() {
        console.log("name: " + name + ", glob: "+glob);
    };
    return closure;
}

var x = createClosure("x");
x();
name = "something";
glob = "Changed";
x();
```

Output:

name: x, glob: Global
name: x, glob: Changed

Figure 2. Example of how JavaScript handles references to global variables in closures.

The second problem with JavaScript's implementation of closures is a result of combining object oriented and functional programming. In JavaScript, and many other object oriented languages, the 'this' variable is used to point to the current context the function is executing in. In normal functions, 'this' refers to the global context (JavaScript stores 'this' as the Window object); however, when inside a method of an object, 'this' refers to the instance of the object the method is being called on. The problem arises when a closure is created inside a method of an object. In this case, the 'this' variable is not stored as part of the scope bound to the closure, so inside a closure, 'this' always refers to the global Window object. An example of this is shown below.

Code:

```
function Object() {
}
Object.prototype.createClosure = function(value, name) {
     console.log(this);
     function closure() {
console.log(this);
     };
     return closure;
};
var obj = new Object();
x = obj.createClosure(5, "x");  // Output will be Object instance x

x();  // Output will be the global scope (Window)
```

Output:

Object {createClosure: function}
Window {...}

*Figure 3. An example of the problem with '**this**' and closures in JavaScript.*

Language	Closures	Functional
C	No	No
Pascal	No	No
C++	Yes	No
Java	Yes	No
Modula-3	Yes	No
Python	Yes	No
Ruby	Yes	No
D (2.0)	Yes	No
Ocaml	Yes	Yes
Erlang	Yes	Yes
Haskell	Yes	Yes

Figure 4: examples of languages that support closure [30]

Does Javascript support higher-order functions like Haskell?

Also known as functionals, higher order functions refer to functions which take a function as a parameter and return a function as a return value [31]. Javascript functions, like in Haskell, can be implemented this way.

First-class functions are also supported in Javascript. Specifically, the language supports passing functions as arguments to other functions, returning them as the values from other functions (as with high-order functions), and assigning them to variables or storing them in data structures [32]. First-class functions, mentioned in the above section on closures, are generally referenced in a programming-specific sense [9]. That is, a language treats functions as values. In comparison, higher-order functions can be defined in a strictly mathematical sense.

A simple and well used example of a higher-order function is “map.” Many languages such as Haskell, Scala, Erlang, Scheme, and Javascript have their own versions of “Map” which takes a function and a list as arguments, then returns a new list formed by applying the function to each element in the list. In order for a language to support map, it must also support higher order functions [33].

As seen in the examples below, Javascript uses higher order functions. However they are implemented differently than in Haskell to accommodate how Javascript deals with closures. For example, the “thisArg” parameter is used to replace the this keyword as the reference to this is not stored within the closure, as mentioned in the section above.

Haskell map function:

```
map                     :: (a->b) -> [a] -> [b]
map f  []               =  []
map f (x:xs)            =  f x : map f xs
```

ex: `map (add 1) [1,2,3]`
`=> [2,3,4]`

Figure 5: example of the Map function in Haskell (partial application of a curried function), function is returned as a value[34].

Javascript Map function:

array.map(callback[, thisArg])

Parameters:
callback: function that produces an element of the new Array from an element of the current one.
thisArg: object to use as **this** keyword when executing callback (optional argument).

Figure 6: example of map function in Javascript, with parameter definitions [35].

“A useful way to think about why you might create functions that return another function is that the arguments to the higher-order function serve to ‘configure’ the behavior of the returned function” [32].
Using Map as an example again, the returned function (i.e. the new list) is “configured” by the parameters (original list, function).

Javascript as a Pure Functional Language: Yes or No?

As defined in the assigned question, any pure functional language must support higher-order functions and implement closure correctly. This is arguably not the case for Javascript. Despite supporting high-order and first-class functions, Javascript fails to correctly implement closures according to the Lambda calculus definition as shown in previous sections.

List Comprehension

In comparing array comprehension in JavaScript to list comprehension in Haskell it is important that the purest definition of list comprehension is readily at hand. After properly defining list comprehension we will begin comparing how it's used in JavaScript and Haskell as specified. Note: JavaScript examples use syntax consistent with FireFox's JavaScript interpreter.

List comprehension is defined as "a syntactic construct available in some programming languages for creating a list based on existing lists. It follows the form of the mathematical set-builder notation (set comprehension) as distinct from the use of map and filter functions" [39]. The set-builder notation is as follows:

$$\{x | x \in \mathbb{U} \wedge \Phi(x)\}$$

Where "$x \in U$" is called the generator, generating a list of element defined by a set "U", and "Φ(x)" is the predicate, subjecting x to some condition defined by Φ(x) [40].

Haskell uses generators very well. Not only can Haskell generate the list for the variable to use but it also allows dependent generators or even user dependent generators.

- Dependent generator example: [(x,y) | x <- [1..3], y <- [x..3]], outputs [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
- User dependent example: factors n = [x | x <- [1..n], n'mod'x == 0] will depend on what the user defines to be n and then list its factors

In JavaScript, however, array comprehensions use iterators instead of generators. This means arrays must be defined before making use of the array comprehension. Since JavaScript does not make use of generators, it obviously does not support dependent arrays or user defined arrays. There are some workarounds in JavaScript such as defining the array using parameters given by the user before the array comprehension and using nested array comprehensions for dependent arrays. Both of these workarounds are clearly not as concise, or elegant, as Haskell's use of generators.

JavaScript does support a form of predicate which is commonly referred to as the 'guard function'. The guard function is typically prefixed with an 'if' and works almost identically as Haskell's predicate. An example of this would be:

ex.1:
```
js> [x for each (x in [0,1,2,3,4,5]) if x > 2]
```

The output for the above example would be [3, 4, 5] and the guard function is 'x > 2'. Haskell using a simple comma to prefix the predicate. An equivalent example in Haskell to the J.S example above would be:

ex.2:
```
s = [x | x <- [0..5], x > 2]
```

Here “x > 2” is the predicate.

Haskell almost exactly implements the set-builder notation shown above. The Haskell code excerpt shown above illustrates this clearly.

JavaScript tries to approximate the set-builder notation, however, JavaScript statements are more wordy. Example 1 shows that JavaScript uses a similar format but strays begins to use more keywords in its syntax. The major difference once again being that JavaScript does not support the generator concept.

JavaScript also has Array methods, part of the new standard library, that act as tools for array comprehension. These methods include Array.map(‘function’) and Array.filter(‘guard function’) used to apply a function to an array and filter an array respectively. JavaScript also has iterator methods such as Array.forEach(‘function’). All of these diverge the style of JavaScript a great deal from both Haskell and traditional set notation and push it towards a function oriented syntax.

A quick example of Array.filter would be:

```
[1,2,3,4,5,6,7,8,9].filter(function(numb) {
            return numb > 5;
            })
```

Which return the list [6, 7, 8, 9].

In terms of theoretical expressivity (defined above) Haskell is more expressive since the set-builder notation is the most compact way of expressing list comprehension statements. JavaScript is less expressive since equivalent array comprehension statements sometimes require longer lines or even multiple lines, and perhaps user defined functions, to obtain the same functionality.

In terms of practical expressivity Haskell, again, is more expressive since it supports the generator concept and allows list to be dependent on each other or even dependent on a variable. With no generator support, JavaScript can do none of these.

Works Cited

[1] [2] Severance, Charles (February 2012). "Java Script: Designing a Language in 10 Days". Computer (IEEE Computer Society) 45 (2): 7–8. doi:10.1109/MC.2012.57. Retrieved 23 March 2013.

[3] (2013, Oct 30). “Popularity” [Online]. Available: http://brendaneich.com/2008/04/popularity/, ECMA brief http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[4] "Usage of JavaScript for Websites." Usage Statistics of JavaScript for Websites, October 2013. http://w3techs.com/technologies/details/cp-javascript/all/all Web. 30 Oct. 2013.

[5] Manger, Jason J. JavaScript Essentials. Pg. 12 Berkeley, CA: Osborne McGraw-Hill, 1996. Print.

[6] Crockford, Douglas. JavaScript: The Good Parts. Beijing: O'Reilly, 2008. Print.

[7](2013, Oct 30). “INDUSTRY LEADERS TO ADVANCE STANDARDIZATION OF NETSCAPE'S JAVASCRIPT AT STANDARDS BODY MEETING” [Online]. Available: http://web.archive.org/web/19981203070212/http://cgi.netscape.com/newsref/pr/newsrelease289.html

[8] (2013, Oct 30). “JavaScript isn’t Scheme” [Online]. Available: http://journal.stuffwithstuff.com/2013/07/18/javascript-isnt-scheme/

[9] (2013, Oct 30). “JavaScript:
The World's Most Misunderstood Programming Language” [Online]. Available: http://www.crockford.com/javascript/javascript.html

[10] (2013, Oct 30). “Comparing Python to Other Languages” [Online]. Available: http://www.python.org/doc/essays/comparisons.html

[11] (2013, Oct 30). “A Short History of JavaScript” [Online]. Available: http://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript

[12] ;c(2013, Oct 30). “Javascript to Scheme Compilation” [Online]. Available: http://www.deinprogramm.de/scheme-2005/10-loitsch/10-loitsch.pdf

[13] (2013, Oct 30). “Douglas Crockford: The JavaScript Programming Language” [Online]. Available: http://www.youtube.com/watch?v=v2ifWcnQs6M

[14] (2013, Oct 30). “Study: Clojure, CoffeeScript and Haskell Are the Most Expressive General-purpose Languages” [Online]. Available: http://www.infoq.com/news/2013/03/Language-Expressiveness

[15] (2013, Oct 30). “The JavaScript Problem” [Online]. Available: http://www.haskell.org/haskellwiki/The_JavaScript_Problem

[16] (2013, Oct 30). “JavaScript: how it all began” [Online]. Available: http://www.2ality.com/2011/03/javascript-how-it-all-began.html

[17] Michaelson, Greg. An Introduction to Functional Programming through Lambda Calculus. Wokingham, England: Addison-Wesley Pub., 1989. Print.

[18] (2013, Oct 30). “THE RISE AND RISE OF JAVASCRIPT” [Online]. Available: http://dannorth.net/2011/12/19/the-rise-and-rise-of-javascript/

[19] (2013, Oct 30). “Programming languages ranked by expressiveness” [Online]. Available: http://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/

[20] (2013, Oct 30). “Usage share of web browsers” [Online]. Available: http://en.wikipedia.org/wiki/Usage_share_of_web_browsers

[21] (2013, Oct 30). “JavaScript” [Online]. Available: http://en.wikipedia.org/wiki/JavaScript

[22] (2013, Oct 30). “JavaScript creator ponders past, future” [Online]. Available: http://www.infoworld.com/d/developer-world/javascript-creator-ponders-past-future-704

[23] (2013, Oct 30). “Functional vs. Object-Oriented JavaScript Development” [Online]. Available: http://msdn.microsoft.com/en-us/magazine/gg476048.aspx

[24] (2013, Oct 30). “Monad (functional programming)” [Online]. Available: http://en.wikipedia.org/wiki/Monad_(functional_programming)

[25] G. Michaelson “An Introduction to Functional Programming Through Lambda Calculus”, Addison Wesley, 1989.

[26] (2013, Oct 30). “Closures” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures

[27] (2013, Oct 30). “Closure (computer science)” [Online]. Available: http://en.wikipedia.org/wiki/Closure_(computer_science)

[28] Richard Cornford (2013, Oct 30). “Javascript Closures” [Online]. Available: http://jibbering.com/faq/notes/closures/

[29] S. Mirghasemi, J. Barton, C. Petitpierre "Naming Anonymous JavaScript Functions," (SPLASH '11) Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, New York, NY, USA, 2011 doi: [10.1145/2048147.2048222]

[30] (2013, Oct 30). ”Functional programming” [Online]. Available: http://www.haskell.org/haskellwiki/Functional_programming

[31] (2013, Oct 30). ”Functional programming” [Online]. Available: http://learnyouahaskell.com/higher-order-functions

[32] M. Fogus “Functional JavaScript”, O'Reilly Media, Inc., June 2013.

[33] (2013, Oct 30) “First-class function” [Online]. Available: http://en.wikipedia.org/wiki/First-class_function

[34] Reuben Thomas, et al. (2013, Oct 30) “A Gentle Introduction to Haskell” [Online]. Available: http://www.haskell.org/tutorial/functions.html

[35] (2013, Oct 30) “Array.prototype.map()” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map?redirectlocale=en-US&redirectslug=JavaScript%2FReference%2FGlobal_Objects%2FArray%2Fmap

[36] (2013, Oct 30) “Lazy JavaScript” [Online]. Available: http://danieltao.com/lazy.js/

[37] (2013, Oct 30) “Node.js” [Online]. Available: http://nodejs.org/

[38] (2013, Oct 30) “Node-lazy” [Online]. Available: https://github.com/pkrumins/node-lazy

[39] (2013, Oct 30) “List Comprehension” [Online]. Available: http://en.wikipedia.org/wiki/List_comprehension#JavaScript_1.7

[40] (2013, Oct 30) “Set-builder Notation” [Online]. Available: http://en.wikipedia.org/wiki/Set-builder_notation

[41](1987, March) Masato Takeichi “Fully Lazy Evaluation of Functional Programs” [Online]. Available: http://repository.dl.itc.u-tokyo.ac.jp/dspace/bitstream/2261/40192/1/K-208424.pdf