Midterm 2
Team F

October 30th, 2013

Jeremy Mohr
Robert Aftias
Lee Gauthier
Tim Ritenour
Mustafa Abualsaud
MacKay McGillivray
Siduo Guan
Mohammed Alghamdi
Francis Harrison

History of JavaScript

Origin of JavaScript

At the creation of the world wide web, in the early 1990's, all web pages were static. A static web page is a web page that delivers its context exactly as stored. Users could only view static information and there was no way for users to interact with the page. The response-action interactiveness within the web page requires a programming language to perform the action. That being said, this language needs to interact immediately without having to reload the page for every action. Around that time, there were two popular web browsers; Netscape Navigator and Internet Explorer. In 1995, The creators of Netscape Navigator realized the need of having a scripting programming language embedded into HTML pages, and this is why they recruited Brendan Eich, to help with the creation of the first programming language for the web browser that can be embedded into HTML pages. Their concept was that to make the browser responsible for interpreting the language's commands without having to compile the code or even install a plug-in. Eich joined the Netscape's browser team and was trying to embed Scheme programming language into web browsers, which was a big influence when he started prototyping for a new programming language. This programming language was created in "10 days in May 1995" [1] and is what is known now as JavaScript. JavaScript's original name, however, was Mocha, chosen by the creator of Netscape Marc Andreessen. It's name was changed to LiveScript in September 1995 and finally to JavaScript upon Netscape receiving a trademark license from Sun. This was somewhat a marketing strategy as Java was very popular at that time.

In an interview on JavaScript's past, Eich was asked what prompted the development of JavaScript:

> *Eich's answer:* "HTML needed a "scripting language", a programming language that was easy to use by amateurs and novices, where the code could be written directly in source form as part of the Web page markup. We aimed to provide a "glue language" for the Web designers and part time programmers who were building Web content from components such as images, plugins, and Java applets. We saw Java as the "component language" used by higher-priced programmers, where the glue programmers -- the Web page designers -- would assemble components and automate their interactions using JS." [2]

Scheme, which is a LISP delicat, was a big influence on JavaScript that led into having closures. Self, an object-oriented programming language based on the concepts of prototypes was another influence that led to having prototypal inheritance in JavaScript. Scheme and Self are sometimes called the parents of JavaScript.

Functional Characteristics - Syntax and Semantics - Lambda Calculus

The Java- prefix in JavaScript's name often leads into confusion towards thinking that JavaScript is a part of Java's platform. JavaScript syntax is very similar to Java and C, but it does not run under a virtual machine like Java nor it needs to be compiled. JavaScript adopts the "curly braces" syntax style like Java and C does.

JavaScript	Java

`for(var i=0; i<100; i++) {`
`  console.log(i);`
`}`
	`for(int i=0; i<100; i++){`
	`  System.out.println(i);`
	`}`

With syntax that appears similar to Java and C, it may appear to some users that JavaScript acts more like an imperative language. Especially with if-condition statements and for loops being very similar to an imperative language. This is not the case, however. JavaScript is considered a functional language. JavaScript supports first-class, a feature that allows passing whole functions as an argument to another. It also supports closure, which is considered one of JavaScript powerful features; a closure is an expression or a function that can have free variables with an environment that binds those variables.

Consider the following example from the Mozilla Developers Network [3],

```
function init() {
    var name = "Mozilla"; // name is a local variable created by init
    function displayName() { // displayName is the inner function, a closure
        alert (name);//displayName uses variable declared in the parent function
    }
    displayName();
}
init()
```

`init()` creates a local variable name and a closure `displayname().displayname()` - the closure is only accessible inside the scope of its parent function, `init()`. Unlike `init()`, display name does not have its own local variable but reuses name, a local variable of its parent.

In JavaScript, block statement do not introduce a scope, i.e. any variable introduced with a block are scoped to the corresponding function or script and the still exist even after the

block executes. Variables in JavaScript have different values in different scopes and therefore, the current value of the variable is dependent on its current state. For example,

```
var x = 1;
{
  var x = 5;
}
alert(x); // outputs 5
```

The code above outputs 5 because the “var x” in the condition exists in the same scope as the “var x” before the condition. Thus JavaScript uses function-based scoping instead although it handles the need for block-scope using closures.

JavaScript implements every function instance as a function object. It implements functions using function expressions and function declarations. In function declaration, a function variable can be defined without requiring variable assignment, they are implemented as standalone constructs and never as conditionally defined nested functions. JavaScript also hoists all declarative functions to the top of the current scope during parsetime because they are not subject to the rules of process.
Function expressions define a function as part of a larger expression syntax such as a variable assignment. They are not evaluated until a variable gets assigned to them i.e they are still undefined even when they are invoked. Unlike function declarations which must always be named, function expressions can either be named or anonymous.[4]

```
// Function Declaration
function isClass(ath){
    return math === true;
}
// named Function Expression
var isClass = function(math){
    return math === true;
}
// anonymous Function Expression
var isClass = function(){
    return math;
}
```

Embedded in JavaScript are some of the underlying concepts of lambda calculus. One of such features is the ability to create anonymous(lambda) functions and assign them variables. These functions can be used to dynamically create other functions and assign them a result which can be used in other parts of the script. Behind these lambda functions is the concept that functions are first-class values that can be manipulated like numbers and strings.

```
var add = function (x, y) {
     return x + y;
}
alert(add(3, 4)); // 7
```

Lambda are often used in JavaScript for:

- Performing operations on the other arguments passed in. For example,

```
var sum = function sum() {
  var result = 0;
  [1, 1, 1].forAll(function add(num) { result += num; });
  return result;
}};
ttest('Lambdas.', function () {
  equal(
    sum(), 3,
    'the result should be 3.'
  );
 });
```

In the .forAll method, the .add(num) function is a lambda. The .forAll method calls .add() for each number in the array. It is important to note that .add() has access to the result of the containing function's scope closure.

- Adding more functionality to existing functions. (Function decorators)
- Returning a function from another function.

Although JavaScript allows for it, it's use of Lambda's differs from the parent concept (Lambda calculus) in some aspects. For example, multi-arguments on a lambda is not allowed in lambda calculus. To compute a multi-argument operation in lambda calculus requires using layers of nested lambda expressions. Implementing multi-argument operations this way can lead to inefficiency in JavaScript. As such, the definition of lambda

calculus is too limited to apply in practical JavaScript applications.[5]

Recursion is a concept that is heavily used in functional programming. Some functional programming solely depends on recursion for repeatedly calling code [6]. Consider the following example to calculate the power of a number.

```
function pwr(base, exp) {
    if (exp===0){
        return 1;
    }else{
        return base * pwr(base, exp-1);
}
```

This function uses recursion for computation. There is still something very imperative in this code, however; the if-else statements. Functional programming tend to use expressions that evaluate to a something.

```
function pwr(base, exp) {
    return (exp===0) ? 1 :
        base * pwr(base, exp-1);
}
```

This is as functional as it can get to make JavaScript look like a functional language. We have replaced the if-else statements with the ? conditional operator. the ? conditional operator returns its first argument if the condition is true, and the second argument if it is false.

As mentioned earlier, JavaScript supports first-class functions. First-class function is one of the functional programming features. this feature allows functions to take functions as an argument, and return functions as arguments. In other words, functions can be treated as variables that can be passed or returned. This is very known in lambda calculus. For example:

```
λx.(λy.x)
```

```
function (x) {
    return function (y){
        return x;
    }}
```

It is important to note that, although JavaScript is considered a functional language, it does not include every possible functional characteristic, like Haskell does. JavaScript does not

have immutable variables or pattern matching features. [7]

Lazy Evaluation

In layman's term, if a programming language has “lazy evaluation” feature, it has the option of only evaluating the value of an expression when needed, instead of as soon as it appears.

Here is a simple example of how lazy evaluation works:

Suppose we have a function (in pseudocode) called “chooseOne” which picks one number from three integer parameters, and print it out on the screen:

```
1:    function chooseOne(x, y, z)
2:        if (x > y)
3:            print x
4:        else if (x < y)
5:            print y
6:        else
7:            print z
8:
9:
10:    chooseOne(1, 2, getMeARandomNumber())
```

In line 10, we called the function “chooseOne”; but notice the 3rd parameter is the return value of a function “getMeARandomNumber()”. Here, the difference between lazy evaluation and strict evaluation surfaces.

Notice the logic in previous code: for the program to reach line 7, “x” and “y” must be unequal; in some cases, we will not need the value of “z”. That means, in this case, a lazy programming language won't evaluate “z”. However in strict evaluation, as long as “getMeARandomNumber()” is passed in as a parameter, the program will evaluate it right away, ignoring the fact that it might not be needed at all.

By allowing lazy evaluation we gain the power to gain the ability to work with infinite lists, as the values can be computed as they are needed. As well depending on how lazy evaluation is used, it can greatly affect (for better or worse) the performance of a program.

Here is a short example of how lazy evaluation helps us in Haskell:

```
magic :: Int -> Int -> [Int]
magic 0 _ = []
magic m n = m : (magic n (m+n))
```

If magic is called with:

```
::> magic 1 1
... (This is not gonna end...)
```

It will expand recursively into an infinite list of fibonacci numbers. This can not be done in a strict language, because the computer can not compute an infinite sequence. With lazy evaluation the whole list is not needed, because the values of ***magic 1 1*** are computed on demand. For example maybe we only want to first 5 elements:

```
::> take 5 $ magic 1 1
[1,1,2,3,5]
```

As JavaScript is strictly evaluated, a function's value is computed as it is called. Luckily, it is still possible to get some of the expressiveness of lazy evaluation by using third party libraries. Here is the same program in JavaScript using the Lazy.js library:

```
var Magic = function(x, y) {
 return function() {
   var prev = x;
   x = y;
   y += prev;
   return prev;
 };
}
var magic = Lazy.generate(Magic(1, 1));
```

Even though we get a "list" with undefined length:

```
::> magic.length();
undefined
```

We can still compute the values on demand:

```
::> magic.take(5).toArray();
[1, 1, 2, 3, 5]
```

In many cases using generators in JavaScript to do stream processing can lead to much better performance. [8]
The big pitfall of lazy evaluation is, when values are mutable and functions have side-effects,

the actual output of the program can become non-deterministic. Therefore, lazy by default only really makes sense for purely functional languages where IO and the order of evaluation does not affect the results of the program, as is the case in Haskell.

Even though lazy evaluation could add expressiveness to a non-pure language (eg. JavaScript), there are usually better abstractions that can be used to get the same expressiveness, as demonstrated by streams. Additionally, rather than extending the core of a language, it is usually less work to extend a language through libraries. For these reasons and probably more, there exists no lazy implementations of JavaScript.

Higher-Order Functions

What is closure:

A closure is a function that is combined with all the local variables where it was declared. These variables are captured by the closure, so you can call the closure from anywhere in the code without any change in functionality. A new closure of the function will have different copies of its environment (the local variables it captures). This can be useful for creating functions that have implicit access to variables without having to pass them in as parameters and to expose them to users. Closures are even becoming increasing popular among non-functional languages as well. The C++11 update to the language added lambda closures, and a pretty significant part of the C++14 update is concerning them as well. They are also getting added in future Java versions.

An example of closure being used in JavaScript is:

```
function environment()
{

   var count = 0;

   function closure()
   {

       count = count + 1;

       return count;
   }

   return closure;
}

var foo = environment();

foo();    // returns 1

foo();    // returns 2

var bar = environment();
bar();    // returns 1
```

In this example foo and bar are different copies of environment, so they both have their own copy of the variable “count”. Since they have their own copy foo and bar will return the value

of count independent of each other.

Higher-Order Functions:

JavaScript supports higher order functions as much as Haskell does, however it does lack some of the syntactic sugar that Haskell has. Some examples of this are function composition which is an operator '.' in Haskell. You can still easily write higher order functions in general, though. An example below is:

```
function compose(f, g)
{
    function ret(x)
    {
        return f(g(x));
    }
    return ret;
}

function foo(x)
{
    return 1 + x;
}

function bar(x)
{
    return x * 2;
}

var foobar = compose(foo, bar);
foobar(10);    // returns 21    (1 *(2 * 10))
```

This could be written more concisely in Haskell as follow:

```
foo x = 1 + x

bar x = x * 2

foobar = foo . bar

foobar(10)
```

Closures in JavaScript:

In JavaScript, functions are objects and each object maintains the scope of the function. The object closure is maintained by binding local variables to a specific instance of a variable instance. When a variable is accessed in a function it checks the parameters/local variables and bound variables. If the variable is not found there it goes up one function block in scope. It keeps going until you reach the global scope. If it's not found at the global scope and this is an assignment rather than an evaluation, JavaScript simply creates a global variable with that name, and errors if you're simply evaluating it. This scope lookup can easily lead to unintended behaviour if you're not careful. The scope in JavaScript is based on functions so certain precautions need to be made when dealing with flow control structures like for loops. [10]

The "let" keyword in JavaScript allows block scoping so that existing variables can be reused and nonexistent variables can be created for life within that block (ex. for loop iterator). However, the let keyword only works in Firefox so it is not platform independent and using it is advised. [9]

List Comprehension

Comprehensions are easy to understand and common language. There are languages that use list comprehensions such as Python and Haskell. These languages are inspired by the mathematical notation of set comprehension.

However, array comprehensions are considered to be convenient, and declarative form for making calculated arrays with a literal syntax that reads naturally. JavaScript 1.7 introduced array comprehension and it looks like the following syntax:

```
var numbers = [1, 2, 3, 4];
var doubled = [i * 2 for (i of numbers) if i%2==0];
```

List Comprehension in Declarative vs Imperative Styles:

JavaScript is fundamentally an imperative language. It is based on calls which affect the state of the program and the data within it. The opposite approach is taken with Haskell, which is functional and avoids heavy use of state. This difference is apparent in the standard way of handling collections of data. In Haskell, lists are the de facto collection of data, whereas in JavaScript, the base object type is an associative array. JavaScript allows calls which arbitrarily access members of objects and change them, whereas Haskell favors declarations which describe relations between data structures with functions. This difference in programming language style influences heavily how programmers express the same idea.

List comprehensions are based on Set-Builder Notation from mathematics, in which some collection is defined based on a base collection with some conditions applied. Haskell list comprehensions mirror this almost exactly, allowing generators or lists to be used for the base collection, and predicates and expressions to impose the conditions. List comprehensions in Haskell are a declarative way of relating collections of data by functions, are thus fit in as a basic feature in the language.

JavaScript's equivalent of a list comprehension is an array comprehension. They allow arrays to be generated providing an input array, predicates, and an expression. While similar in style to list comprehensions and set builder notation, the output of array comprehension is different in that it is not immutable nor is it evaluated lazily. Ultimately, it is syntactic sugar for iterating over mutable input and constructing a mutable output array. While it is syntactically similar to Haskell, the JavaScript implementation loses the mathematical purity of the Haskell implementation.

While both languages could create the equivalent output of their comprehension using

alternative syntax, list and array comprehensions allow an alternative, often simpler, syntax in which programmers can express their code. Though the implementations for array and list comprehensions reflect the difference in style between the declarative Haskell and the imperative JavaScript, both comprehensions provide a clean alternative syntax for expressing certain ideas.

List/Array Comprehension in Haskell and JavaScript:

Array comprehension is supported in JavaScript as of version 1.7. The array comprehension can support many features of Haskell's list comprehension; however, JavaScript requires very different implementation. Implementing a list in haskell can be done quite simply. For example, creating a list of integers from 1 to 9 multiplied by three can be done as follows:

```
[x * 3 | x <- [1, 2 .. 9]][x * 3 | x <- [1, 2 .. 9]]
```

[11]

Achieving the same list as an array in JavaScript can be done a few ways:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
var tripled = [i * 3 for (i of numbers)];
```

[12]

Another way would be to use the map() operation:

```
var tripled = numbers.map(function(i){return i * 3;});
```

In Haskell filtering out elements of a list can be done by adding a predicate to the comprehension. For example filtering the even numbers from a list of integers 1 to 9 is done by:

```
[x | x <- [1, 2 .. 9], x `mod` 2 == 0]
```

This can be implemented in JavaScript a number of ways:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
var evens = [i for (i of numbers) if (i % 2 === 0)];
```

It can also be done by using the filter() operation:

```
var evens = numbers.filter(function (i){return i % 2 === 0;});
```

In both Haskell and JavaScript the operation and the predicate can be added into a single implementation. In Haskell the combined operation of tripling the integers from 1 to 9 and filtering the even numbers can be done as follows:

```
  [x * 3 | x <- [1, 2 .. 9], x `mod` 2 == 0]
```

Implementing this list as an array in JavaScript can be done as follows:

```
   var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
   var tripledEven = [i * 3 for (i of numbers) if (i % 2 === 0)];
```

While the resulting lists and arrays are equivalent the implementations differ vastly. List comprehension in Haskell is straightforward and mathematically intuitive. However array comprehension in JavaScript requires operations from the library, the use of loops, and additional declarations to achieve the same result. The way JavaScript handles array comprehension is very similar to that used by Ecma International's standardized language, ECMAScript.

Generators and Predicates for both Haskell and JavaScript:

To follow up with the previous Haskell example, explaining more on generators:

```
[x * 3 | x <- [1, 2 .. 9], x `mod` 2 == 0]
```

For x <- [1, 2, ..., 9], this is the generator and it has 9 slots waiting for the value to be assigned in the list. So, x*3 is an expression that holds the value of the input. And it will be ready to take another input once the generator assigned the specified index with the previous X for the expression. But, before assigning the values in the list, the predicate will filter the values to allow what value meet the condition and what does not. Therefore, as seen from the example, it uses a simple statement to filter the values.

On the other hand, to follow up with a previous JavaScript example,

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
var tripledEven = [i * 3 for (i of numbers) if (i % 2 === 0)];
```

here we declared all a single array that has 9 slots numbered with 1 to 9, and this acts as the generator. Then, the second line will act as the expression for which all values of i will be assigned in the array. Also, it is the predicate where all values will go through this filter and it will go under a loop method. Nevertheless, as seen from the example code, the statement seem to be not as simple as the Haskell's predicate. [13]

References:

[1] A Short History of JavaScript (June 2012). W3C [Online] Viewed 2013 October 30. Available: http://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript

[2] The A-Z of Programming Languages: JavaScript (July 2008). ComputerWorld [Online] Viewed 2013 October 30. Available http://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/

[3] Closures (Octobar 2013). Mozilla Developer Network [Online] Viewed 2013 October 30. Available https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures

[4] Function Declarations vs. Function Expressions (July 2010). JavaScript, JavaScript… [Online] Viewed 2013 October 30 http://javascriptweblog.wordpress.com/2010/07/06/function-declarations-vs-function-expressions/

[5] Chapter 3. Functions. oscon [Online] Viewed 2013 October 30 http://chimera.labs.oreilly.com/books/1234000000262/ch03.html#lambdas

[6] Recursion (computer science) (Octobar 2013). Wikipedia [Online] Viewed 2013 October 30. http://en.wikipedia.org/wiki/Recursion_%28computer_science%29

[7] Functional Programming in JavaScript (September 2012). DailyJS [Online] Viewed 2013 October 30. http://dailyjs.com/2012/09/14/functional-programming/

[8] Like Underscore, but lazier () Daniel Tao [Online] Viewed 2013 October 30. Available: http://danieltao.com/lazy.js/

[9] The let keyword in the javaScript [Online] Viewed 2013 October 30 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let?redirectloc_ale=en-US&redirectslug=JavaScript%2FReference%2FStatements%2Flet

[10] Explaining javascript: explained [Online] Viewed 2013 October 30 http://benmccormick.org/benmccormick/blog/2013/01/08/javascript-explained-closures/

[11] List Comprehension (September 2007). The Haskell Programing Language [Online] Viewed 2013 October 30. Available: http://www.haskell.org/haskellwiki/List_comprehension

[12] Predefined Core Objects (October 2013). Mozilla Developer Network [Online] Viewed 2013 October 30. Available:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Predefined_Core_Objects#Array_comprehensions

[13] Iterator and generators (May 2013). Mozilla Developer Network [Online] Viewed 2013 October 30. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators