**Q1) History**

**It is interesting to learn that there is a strong connection between Lambda Calculus and Javascript. Javascript is similar to Python in many ways; it uses dynamic typing and is mostly interpreted. Try rediscover Javascript as much as you can, as related to its origin, its "functional" characteristics and its expressiveness as a higher-order functional language. Please concentrate on its history as far as its connection to Lambda Calculus and Functional Programming, and ignore other object-oriented features.**

JavaScript was originally created by Brendan Eich in 1995, coincidentally the same year that lambda calculus inventor Alonzo Church died. The language was developed by Netscape, who wanted a lightweight interpreted language to complement Java. JavaScript was originally developed with the name Mocha before it was changed to LiveScript during beta-testing and finally JavaScript by the time of its release.

JavaScript is often thought to be related to Java. This is a common misconception. Java doesn’t support functional programming, where as JavaScript does, because it contains features based on Scheme. The name JavaScript was chosen as a marketing ploy to tie in with Java, which at the time was a new and popular programming language.

JavaScript is embedded within the web browser and is connected through interfaces called Document Object Models (DOM) to applications. One of the biggest uses of web-based JavaScript is to create smaller functions that are embedded into HTML pages and interact with the DOM of the browser to perform various tasks that wouldn’t be possible in static HTML.

JavaScript has the frameworks to simplify object-oriented JavaScript as well as function-oriented JavaScript. JavaScript supports both paradigms and leaves which one to use up to the user. While JavaScript supports both paradigms its strength and weakness is that is doesn’t focus on either and makes the user decide for themselves.

While JavaScript isn’t truly a functional language, it does support some constructs that are typical of a functional language. By using the existing constructs extensively JavaScript can do good functional programming. Anonymous functions are the building blocks of functional programming in JavaScript. Anonymous functions are direct offshoots of Lambda Calculus. In JavaScript anonymous functions are known as function expressions. The difference between a regular function and an anonymous function is in the name, in JavaScript it isn’t necessary to name functions especially if they’re being used as a value to be passed around.

JavaScript version 1.8 was released in June 2008. In this version expression closures were added to JavaScript. Giving it a shorthand method for writing simple functions. This gives the language a notation similar to typical Lambda notation.

**Question 2) (Lazy Evaluation) Haskell is a non-strict, fully-lazy functional language; it can support stream processing using infinite lists without any special operations. For example, ML is another functional language which is not lazy; Scheme is another; they both use call-by-value semantics. There are other techniques of doing stream processing, e.g., lazy-cons or cons-stream in Scheme, or delayed evaluation in ML. Similarly, Javascript uses call-by-value and is not lazy. Discuss how lazy evaluation can improve and enhance the expressiveness of Javascript, and whether there is any lazy-implementation of Javascript.**

Call-by-value is the most common evaluation method. In call-by-value, expressions are evaluated before being passed into functions, whose value is then assigned to a local variable within the function scope. For example:

function f1(var1) {

console.log(var1)

}

function f2(var2) {

return "alert ".concat(var2);

}

f1(f2("poop"));

in call-by-value, the result of f2 is determined (alert + var2) before calling f1. Lazy evaluation, on the other hand, involves passing a reference to the argument to be evaluated when it’s needed. In JavaScript, if we were to instead write f1(f2) and, inside f1 write f2(), the function would have only been evaluated when it was needed.

In operations where certain sections aren’t necessarily guaranteed to be executed, lazy evaluation helps to optimize code by only evaluating things that are needed. For example, in a “lazy” language, if you were to write:

if a then b else c

b would only be evaluated if a were true, else only c would be evaluated. Thus, instead of three potentially expensive operations, there are only two. In a “call-by-value” language, all three components would be evaluated before the expression was executed.

Lazy evaluation is a method in which expressions are evaluated at the last possible moment, often for the purpose of optimizing an algorithm. At the time of evaluation, the data used in a calculation is required to be available and meaningful. It may be difficult to implement and counter-intuitive, which can make debugging a difficult process. Lazy evaluation is very beneficial for expressions that are expensive, particularly recursive definitions. For these, data is evaluated as soon as it is consumed for recursive functions, and there is no need to prepare or hold on to each level produced.

For languages that support state change, lazy evaluation is especially tricky to implement. The timing of lazy evaluation must fall in the window of opportunity for an expression to be properly evaluated. If it is evaluated too late, the data may no longer be valid, and if it is evaluated too early, the data may not be available by the time it is required.

Were JavaScript to support lazy evaluation natively, programmers could write complex statements or blocks of code without boilerplate or complicated code. A good example of this is website pagination. Passing a “page” function a list of all the posts on a website without lazy evaluation could be costly - if the website has a lot of posts, evaluating the list before passing it into the “page” function could greatly impact performance. Since the “page” function only needs a small subset of the list, passing in a reference and allowing the paging function to deal with only the section of the list that it needs.

Lazy.js is a JavaScript library which implements lazy evaluation. It allows for operations to occur in a sequence without calculating the results after each step. Functions return “sequence objects” with an “each” method. Only when this method is called is the function evaluated. This allows functions to be chained without evaluating them until specifically told to. This allows for many “lazy” features in Javascript including support for infinite series’.

Another lazy implementation is wu.js. In the words of it’s creator, it’s “a lazy, functional JavaScript library that ain’t nuthin’ ta f\*ck wit.” It includes higher order functions including pattern matching similar to the method used in Haskell. It also supports infinite lists by yielding only a small part of the sequence at a time.

**Question 3)**

**Q3) (Higher-Order Functions and Closure) Any pure functional language must support higher-order**

**functions and implement closure correctly.**

**What is a closure?**

In order to define a closure we must first understand higher-order functions and nested functions.

A higher-order function does at least one of the following:

* take one or more functions as parameters
* returns or outputs a function

For example:

**function** lazyAdd(x,y,func) {

var sum = func(x,y);

}

**function** add(x, y){

return x+y;

}

lazyAdd(1,2,add);

Nested Functions:

Some programming languages such as Javascript and Haskell support nested functions, however, languages like C and C++ do not.

In JavaScript, if you use the ‘function’ keyword inside of another function, you are creating a closure.

For example:

**function** numbers() {

var num = 1 // local variable

**function** incrementnum() {

num++; //now num is 2

}

**function** squarenum() {

num = num \* num; //now num is 4

}

return num; //returns 4

}

var x = numbers(); //x = 4

* In this example, the function numbers() creates a local variable ‘num’ and the functions called incrementnum() and squarenum(), which are inner functions (*closures*) - these functions are defined inside numbers(), and are only available within the body of that function. The functions incrementnum() and squarenum() do not contain any local variables of their own and use the variable ‘num’ which was defined inside their parent function numbers() and replaces the variable ‘num’ with the functions appropriate operations.
* This is a **closure** because the variable ‘num’ is used and returned from the inner functions incrementnum() and squarenum() and kept ‘*alive*’, i.e. can continue to be used and changed further after the return.
* This is also an example of *lexical* *scoping*: in JavaScript, the scope of a variable is defined by its location within the source code and nested functions have access to variables declared in their outer scope.

In essence, closures are functions that refer to **free variables**. A free variable is avariable referred to in afunction that is not alocal variable or anargument of that function. Variables from the parent function of the closure remain bound from the parent's scope. Here is an example using Lambda Calculus and an open lambda term to demonstrate:

* λx. → introduces a new scope which lasts for the length of the lambda expression, and x is a local variable in that scope.
* λx.x → the x defined is now bound to the local variable x in the scope.
* λx.x y → y is a free variable. We can pair this open lambda term with an environment that maps variables to values, that is:
* λx.x y [y = 2] is a closure because y is a free variable which is now assigned.

A closure can also be defined as an open lambda term paired with an environment that gives values to all of its free variables.

Javascript also implements closures using activation objects. 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. Here is another example of closure created by returning a function object and using activation objects in JavaScript:

var x = 1; //global variables

var y = 2;

**function** add(y)

{

return **function** doAlert()

{

alert(x + y);

}

{

var z = add(5)

z();

* When we ask for the value of *x* inside function doAlert(), JavaScript looks at its 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.
* When a function like ‘add’ 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.
* Thus, with this example, the last code snippet would display “6” instead of the “3” that it could be confused with. We can clearly see this as it would use the ‘y’ in its nearest lexical scope which would be the “5” in the argument of add() instead of the “2” in the global variable.

Thanks to closures, JavaScript functions always execute with their lexical scope!

References:

<http://c2.com/cgi/wiki?HigherOrderFunction>

<http://odetocode.com/Blogs/scott/archive/2007/07/10/closure-on-javascript-closures.aspx>

<http://math.stackexchange.com/questions/118298/lambda-calc-bound-and-free-variables>

<http://stackoverflow.com/questions/111102/how-do-javascript-closures-work>

**Question 4)**

**(List Comprehension)**

**Historically, it comes from programming languages that support sets as a built-in data type. Javascript has some form of array comprehension. List comprehension in Haskell requires support of generators and predicates. Discuss whether similar features are supported in Javascript, and compare its array comprehension against the Haskell's style of list comprehension. Are they the same? Which is more expressive?**

**In Javascript:**

Array comprehensions were introduced in JavaScript 1.7. Array comprehensions in Javascript act in a similar way to list comprehensions in Python. They construct a new array that is a modified version of another. This new array is modified by the predicate which is set in the array comprehension.

**Example 1**:

*var numbers = [1, 2, 3, 4];  
var doubled = [i \* 2 for (i of numbers)];*

In Example 1, *numbers* is the the input array for the comprehension *doubled*. *numbers* is modified by the expression *i\*2*, and the returned array would be *[2, 4, 6, 8]*. This comprehension acts like the map() function.

A filter() comprehension doesn’t modify the elements of the input array. Instead, using a specified predicate, it determines which elements from the original array will remain in the return array.

**Example 2:**

*var numbers = [1, 2, 3, 21, 22, 30];  
var evens = [i for (i of numbers) if (i % 2 === 0)];*

Example 2 returns an array that contains all the even numbers from *numbers*, *[2,22,30]*.

These two concepts can also be combined so that the map function is only applied to elements of the array that pass through the filter.

**Example 3:**

*var numbers =[1, 2, 3, 21, 22, 30];  
var doubledEvens = [i \* 2 for (i of numbers) if (i % 2 === 0)];*

Example 3 combines the map() and filter() type comprehensions.

It should be noted that a javascript array comprehension does not need to have an explicit array passed to it. It may also use an iterator or a generator to be modified by the predicate. In this way you can create an array that contains a large amount of elements without having to declare them explicitly.

There is also a separate language called CoffeeScript which has the same semantics as Javascript, but it with a neater syntax. Using CoffeeScript allows for nesting array comprehensions in a way that is easier to read and write.

**In Haskell:**

There are 3 major types of list comprehension**:**

**1. One may have multiple generators, separated by commas.**

*[(i,j) | i <- [1,2], j <- [1..4]]*

This yields the result *[(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4)]*.

Note how each successive generator refines the results of the previous generator. Thus, if the second list is infinite, one will never reach the second element of the first list.

For example,

*take 10 [ (i,j) | i <- [1,2], j <- [1..]]*

This yields*[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10)].*

In such a situation, a nested sequence of list comprehensions may be appropriate. For example,

*take 5 [[ (i,j) | i <- [1,2]] | j <- [1..]]*

This yields *[[(1,1),(2,1)], [(1,2),(2,2)], [(1,3),(2,3)], [(1,4),(2,4)], [(1,5),(2,5)]].*

**2. One can also provide boolean guards (predicates).**

For example,

*take 10 [ (i,j) | i <- [1..], j <- [1..i-1], gcd i j == 1 ]*

This yields *[(2,1),(3,1),(3,2),(4,1),(4,3),(5,1),(5,2),(5,3),(5,4),(6,1)].*

**3. One can also make local *let* declarations.**

For example,

*take 10 [ (i,j) | i <- [1..], let k = i\*i, j <- [1..k]]*

This yields *[(1,1),(2,1),(2,2),(2,3),(2,4),(3,1),(3,2),(3,3),(3,4),(3,5)].*

Haskell reads and interprets its generators and predicates from left to right. Changing their order may change the output of the comprehension. The ability to set more than one predicate also allows one to filter the list as many ways as desired all in one comprehension.

**Comparison:**

Both Haskell and Javascript use predicates to filter() their lists/arrays. Haskell has the benefit of being able to use multiple predicates to further filter the elements in a list. In Javascript, one would have to write a new comprehension to do so. If one wants to write nested comprehensions as in Python, using CoffeeScript is the easiest, and neatest way to do so.

Haskell is also unique in the way that it reads its generators and predicates from left to right. This means switching the order of your predicates, for example, could change the output of your comprehension. With Javascript you are forced to write the generator, and then the predicate of your choosing.

Both Haskell and Javascript allow you to apply functions to each element of a list/array and then return the result in the return array. These are easy ways to implement the map() function in one line.

While the two languages are very similar in their use of comprehensions, it is Haskell that is more expressive. It is much easier to manipulate and filter the elements of a list in many ways all in one comprehension.

**References:**

**http://www.javascript-help.com/javascript.html**

**http://msdn.microsoft.com/en-us/magazine/gg476048.aspx**

**http://www.haskell.org/haskellwiki/List\_comprehens**ion

**http://www.javascriptkit.com/javatutors/closures.shtm**[**l**](http://www.javascriptkit.com/javatutors/closures.shtml)

**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Predefined\_Core\_Objects**

**http://wiki.ecmascript.org/doku.php?id=harmony:array\_comprehensions**

**http://c2.com/cgi/wiki?LazyEvaluation**

**http://philosopherdeveloper.com/posts/introducing-lazy-js.html**

**http://danieltao.com/lazy.js/**

**http://fitzgen.github.io/wu.js/**