|  |  |  |
| --- | --- | --- |
| CSC 330 Test 2 JavaScript & Lambda Calculus | October 30  2013 | |
| *By: Philip Snead, Samuel Willis, Warren Spencer, Robert Martin, David Bell, Keiran Reilly, Kevin Dahl, Lucas Main, Jordan Casoli* | |  |

Contents

[History and Development 2](#_Toc370936025)

[Fully Lazy Evaluation 4](#_Toc370936026)

[Higher Order Functions and Closure 6](#_Toc370936027)

[List Comprehension 11](#_Toc370936028)

[References 14](#_Toc370936029)

# History and Development

The first form of JavaScript was made in 10 days in May 1995, as a more developer-friendly language for Netscape. It was developed by Brendan Eich, who was hired by Netscape for this purpose. During development, it was referred to as “Mocha”, and later “LiveScript”. At the time, web browsers were considered to be a kind of client-server OS. Netscape saw a need for an interpreted language to complement Java, which was then the primary language for Netscape development. JavaScript was intended to be more accessible than Java, to attract more developers to Netscape. It was designed to be imperative, structured, prototype based object oriented, and “first class” functional, meaning that functions are treated as objects. It was also designed to support both implicit and explicit delegation of functions, as well as lambda calculus based expressions, regular expressions, and run time evaluation. After receiving the appropriate trademark from Sun, it was officially renamed JavaScript in version 2.0B3

Brendan Eich had a bachelors in computer science and mathematics, as well as a masters in computer science, when he began his career at Silicon Graphics in 1986 working on operating systems and network code. He worked there for 7 years before moving on to MacroUnity Systems Engineering to work on their microkernel and digital signal processing code. Three years later, he was hired by Netscape Communications Corporation to create what would eventually become JavaScript.

In 1996-1997 it was submitted to the European Computer Manufacturers Association (ECMA) to create a standardized specification, so that it could be implemented consistently across browsers. This would lead to the official release of ECMA-262 Ed. 1. The standard is officially named ECMAScript, of which JavaScript is an implementation. ActionScript 3 is another implementation of ECMAScript. ECMAScript standards 2 and 3 were released in 1998 and 1999 respectively, the latter of which is the basis for modern JavaScript. In 2000, there was a project led by Waldemar Horwat to implement a JavaScript 2, or “ES4”. Microsoft initially participated in its development, though over time stopped cooperating with the other developers. Without the official support of Internet Explorer (which then comprised a vast majority of browser share), production on JS2 was ended. While JavaScript itself has not undergone significant change since its early standardization, other software libraries such as AJAX and JQuery have been created to further extend JavaScript’s functionality.

# Fully Lazy Evaluation

With lazy evaluation, a value passed to a function is not evaluated until the value is actually used. This means that if a function is passed as an argument to another function and is not used, the function will not be evaluated. This can reduce computation time if the function being passed requires complex, or even infinite, calculations and prevent errors if the computation would otherwise cause an error.

|  |
| --- |
| function f(x,y){  return x; } f(2,1/0); |

This code will not cause an error with lazy evaluation, since the 1/0 is never used. However, with an eager evaluation scheme, the 2 and 1/0 would be evaluated before the function call, causing an error.

SYNTACTIC SUGAR

Javascript does not implement a lazy evaluation but it can mimic its ideas. For example, consider the append function from class (a higher order function), which recursively updates the tail of the list to be appended to the head of the list. The head of the list is always a concrete solution, however the tail is defined as a function call which is never concretely defined or evaluated. This allows for the lazy aspect. In the following JavaScript example we generate an infinite list of the natural numbers by using the same principles as functional programming.

|  |
| --- |
| function ones() {   return new Stream( 1, ones );  }  function naturalNumbers() {   return new Stream(1, function () {   return ones().add( naturalNumbers() );   }   );  } |

We see there are 2 functions, “ones” and “naturalNumbers”. “ones” keeps track of the entire list while “naturalNumbers” appends the next natural numbers recursively.

The previous JavaScript code can be expressed using functional language syntax as well:

|  |
| --- |
| naturalNumbers [1] = [1] naturalNumbers [x:xs] = x : naturalNumbers [x+1:xs] |

This example is not an implementation of lazy evaluation but merely attempts to mimic its features.

While these features can be emulated in some cases with clever coding, it still requires function-specific implementations that must be re-written for each new case. Javascript could be improved in power and expressiveness by including it in the core of the language, allowing lazy evaluation to be used universally.

# Higher Order Functions and Closure

In terms of traditional programming, a closure can be thought of as a special object that binds a function with any exterior environment variables that it uses. Closure requires nested functions and is created when an inner function is made accessible outside of the outer containing function. A possible technique for achieving closures is assigning a reference to the inner function to a global variable or a property of a global object.

Consider the following JavaScript function example:

|  |
| --- |
| **Example 1:** |
| function foo() {  var name = "Closure in Action";  function displayName() {   document.write(name);   }  return displayName();  } foo(); |

In the example above there are two functions declared: Function foo() and function displayName(). Here, displayName() is not self contained, or “closed”, because it relies on the the variable name which is declared outside the functions scope. In creating a closure, the variable nameis stored and linked to the displayName() function, allowing all calls to displayName() to use the value of name.

This is important because displayName() can now modify or manipulate the name variable. Due to JavaScripts use of references, a change to name in the displayName() function will be reflected to the appropriate environment variables. Below is an example of this:

|  |
| --- |
| **Example 2:** |
| var Counter = ( function() {  var privateCounter = 0;  function changeBy(val) { privateCounter += val; }  return {  increment: function() { changeBy(1); },  decrement: function() { changeBy(-1); },  value: function() { return privateCounter; }  };  })(); |

Using the above example, the increment, decrement, and value functions are all closures that share the same environment. Because of this, the following Counter calls will all modify the same variable, much in the way a class might work in Java.

|  |
| --- |
| alert(Counter.value()); */\* Alerts 0 \*/* Counter.increment(); Counter.increment(); alert(Counter.value()); */\* Alerts 2 \*/* Counter.decrement(); alert(Counter.value()); */\* Alerts 1 \*/* |

Like Java, JavaScript uses references to perform automatic garbage collection (freeing of memory). This means that JavaScript tracks how many active references an object variable has. When an object variable no longer has active references within the program, then it can be garbage collected.

Looking back at Example 1, when the displayName() function is defined, it is given a reference to the variable name. This prevents the variable namefrom being garbage collected. The function is also given a reference to the scope chain itself, which locks the scope chain and prevents it from being removed. This allows continual reference to the variables in the execution context of displayName().

Often closures will rely on parameters passed to a function which is higher in the scope chain. Consider example 3:

|  |
| --- |
| **Example 3:** |
| function exampleClosureForm(arg1, arg2) {  var localVar = 10;  function exampleReturned(innerArg) {  return ((arg1 + arg2)/(innerArg + localVar));  }  */\* return a reference to the inner function defined as -  exampleReturned -:-  \*/*  return exampleReturned; }  var closure1= exampleClosureForm(2, 4); var closure2= exampleClosureForm(12, 24); |

In example 3, two closures are formed on the same function. Each closure is created independently, meaning that each environment is referenced as an independant scope chain with distinct variables. For example, the scope chain for closure1 is {exampleReturned1, exampleClosureForm1, closure1} while the scope chain for closure2 is {exampleReturned2, exampleClosureForm2, closure2}. Whilst resolving the values of identifiers, the program precedes through the scope chain attempting to match the identifier with a property of the same name. In example 3, if reference is made to arg1 by closure1 the scope chain is parsed starting with exampleReturned1. Since exampleReturned1 does not contain the value for arg1, the next object on the scope chain is checked. exampleClosureForm1 does contain an arg1 and so this is assumed to be the arg1 which is being referenced.

Both JavaScript and Haskell support higher-order functions, but they do so in slightly different ways. JavaScript functions have the ability to accept more than one parameter at a time, while functions in Haskell only take in one parameter as input. Haskell uses a technique called currying as a way to simulate allowing multiple parameters. Currying also allows for application of functions, while also allowing for the transformation of functions on the fly. Functions can be applied fully or in part. In part meaning that it is possible to call a function without specifying all the parameters required for the function to return its intended value. Instead the function will return another function that takes the remaining parameters as input. Javascript also provides this functionality, but is slightly more difficult to implement. In JavaScript, it is necessary to explicitly handle partial application of functions. Here is a simple example:

|  |
| --- |
| */\* a curried add  accepts partial list of arguments \*/* function add(x, y) {  if (typeof y === "undefined") { *// partial*  return function (y) {  return x + y;  };  }  *// full application*  return x + y; } *// test*   typeof add(5); *// "function"* add(3)(4); *// 7* *// create and store a new function* var add2000 = add(2000); add2000(10); *// 2010* |

Notice that if argument y is not specified in the parameters, the partial application will be handled inside the add() function. It is possible to handle this in a more general fashion by defining a function that transforms any function into a new one that accepts partial parameters. The code for this is not all that complicated. Below is an example of how it could be used. Assume the function called curry to handle partial function application.

|  |
| --- |
| *// a normal function* function add(x, y) {  return x + y; } *// curry a function to get a new function* var newadd = curry(add, 5); newadd(4); *// 9* *// another option -- call the new function directly* curry(add, 6)(7); *// 13* |

In Haskell, a space between two arguments is used to imply function application. Consider the following Haskell function:

|  |
| --- |
| multThree :: (*Num* a) => a -> a -> a -> a  multThree x y z = x \* y \* z |

When multThree 3 4 5 is called, 3 is applied to multThree first, because the values 3, 4, 5 are separated by spaces. This creates a function that takes one parameter and returns a function. The new function takes in the 4 and multiplies it with the 3 , and then returns a function that will take in a final parameter and multiply it by 12. The final parameter, 5, is then applied to the function and the final result is an integer. Notice that this function's type could also be written as:

|  |
| --- |
| multThree::(*Num* a)=>a->(a->(a -> a)) |

This is an example of a curried function in Haskell. In conclusion, both Haskell and JavaScript support higher order funtions. They do so in a very similar fashion and allow for almost all of the same functionality.

# List Comprehension

Array comprehensions are a powerful tool in JavaScript. Using array comprehensions, programmers are able to create dynamic arrays using the values of other iterables. For the most part the functionality they provide could otherwise be achieved using loops, however using array comprehensions leads to far more concise and readable code. For example, take the following two equivalent fragments of JavaScript code:

|  |
| --- |
| var initialArray = [1, 2, 3]; var doubledArray = [i \* 2 for (i of intArray)]; |

|  |
| --- |
| var initialArray = [1, 2, 3]; var doubledArray = []; for (i=0; i<3; i++){  doubledArray.push(a[i] \* 2); } |

Both will populate the array, doubledArray with the values [2, 4, 6], however in this example using a list comprehension cuts the lines of code required in half.

From the above example it is apparent JavaScripts’ Array comprehensions provide an easy shortcut to produce an array based on the content of another array. The input, however, does not necessarily need to be an array as one can use iterators, generators, or even strings. The following is the syntax for an array comprehension in JavaScript:

|  |
| --- |
| [ *Expression0* for ( *LHSExpression1* of/in *Expression1*) … for ( *LHSExpressionn* of/in *Expressionn* ) if *( Predicate )optional* ]; |

Where the resulting array consists of all values computed by *Expression0*. Where each *Expression0* is resultant from the values given by each unique permutation of the LHS expressions satisfying the volitional expression ***(*** *Predicate* ***)***.

The square brackets in an array comprehension introduce an implicit block and variables are treated as if they have been declared using a let. This means that variables introduced within the array comprehension are not available outside of the comprehension. This syntax is similar to Haskells list comprehension, the main difference being in the ordering of the inputs and the use of keywords. Haskell’s list comprehension is a syntactic sugar that has the following form:

|  |
| --- |
| **[** *Expression* **|** *q1* ***,*** *q2* ***, . . . ,*** *qn* **]** |

Where each qualifier *qi* is one of:

1. **Generator**: p ← e. Where p is a pattern of type T and e an expression of type [T]
2. **Guard**: an arbitrary Boolean expression
3. **Local Binding**: new definitions provided for use in the generated Expression or the following guards and generators.

Haskell’s list comprehensions do not impose any restrictions on the ordering of qualifiers, which are read sequentially from left to right. In JavaScript this is not possible, and the single optional predicate comes after the last generator.

The functionality provided by JavaScript array comprehensions is similar to Haskells list comprehensions. Both allow the use of iterables or generators as qualifiers, predicates to test against the iterated values. We can use both to compute, for example, cross products of sequences. In JavaScript:

|  |
| --- |
| var myArray = [[x, y] for (x of [1, 2]) for (y of [4, 5, 6]) if (y < 6)]; |

and in Haskell:

|  |
| --- |
| myArray ← [(x, y) | x ← [1, 2], y ← [4, 5, 6], y < 6]; |

both yield [(1, 4), (1, 5), (2, 4), (2, 5)] as myArray.

It is possible in both Haskell and JavaScript to produce infinite generators. In Haskell this is quite simple to do and does not require too much extra thought. In JavaScript it requires the creation of a separate generator function. There is an example of a JavaScript infinite generator that produces the Natural numbers in the Lazy Evaluation section.

Haskell has a less restrictive syntax and allows for ordering its qualifiers as desired, while JavaScript enforces the generators to appear before the predicate. As well, in Haskell, one can use the assignment operator “let”to assign specific values to variables within the list comprehension. Note that the “let”operator in Haskell is not the same as the “let”in JavaScript. However, when comparing JavaScripts and Haskells keywords, JavaScripts use of for, of / in and if is much clearer than Haskells use of |, followed by any number of generator or predicate functions. Despite this, Haskell’s convention of following generators by a predicate specific to that generator makes it much clearer where each predicate is used.

# References

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Predefined\_Core\_Objects
* http://gafter.blogspot.ca/2007/03/on-expressive-power-of-programming.html
* http://www.haskell.org/onlinereport/exps.html
* http://ghc.io/
* http://www.haskell.org/haskellwiki/List\_comprehension
* http://www.JavaScriptkit.com/javatutors/closures.shtml
* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures
* http://jibbering.com/faq/notes/closures/
* http://learnyouahaskell.com/higher-order-functions
* http://philosopherdeveloper.com/posts/introducing-lazy-js.html
* http://www.drdobbs.com/open-source/currying-and-partial-functions-in-javasc/231001821?pgno=2
* http://www.drdobbs.com/open-source/currying-and-partial-functions-in-javasc/231001821?pgno=2