Advanced Topics in Programming Languages **Spring Semester, 2012**

Lecture 2: March 13, 2012

Lecturer: Prof. Mooly Sagiv *Scribe: Omri Gindi and Roi Becker*

JavaScript

2.1 General

JavaScript is a language with weak and dynamic typing, and functions as first-class objects. It is multi paradigm, in a sense that it supports many styles of programming, including functional, object oriented and imperative.

JavaScript was first developed by Brendan Eich from Netscape in 1995. It was later formalized by the ECMAScript, and is now implemented by all popular browsers.

Its main use is for building web pages to be displayed in web browsers. The script itself is embedded inside an HTML page, for example, and thus the web browser can run it. It is usually done in the following manner:

```
<HTML>
<HEAD>
    <SCRIPT LANGUAGE="JavaScript">
        function processOrder() {
            // JavaScript statements go here
        }
    </SCRIPT>
</HEAD>
<BODY>
    <FORM NAME="myForm">
     <INPUT TYPE="button" NAME="processOrder" VALUE="Click to process"
onClick="processOrder();">
</HTML>
```

In this simple HTML document, we declare the JavaScript function in the relevant tag, and we can later use it as a handler for button clicks.

JavaScript was designed to allow non programmers an easy way to develop webpages simply by copying existing JavaScript code, and making the needed changes into it. It is very good in tolerating small errors as well (for example semicolons are optional) and very simple and quick to learn and develop. It being weakly and dynamically typed makes code more flexible and readable (with the price of being less safe). Also, since it runs on the end user's computer, it can do the simple things much faster than it would have done if it was run on a web server. However, these advantages come with a very expensive price, security. The fact that types are not checked in a stage prior to runtime can be used for malicious purposes. Many people disable JavaScript from running on their browsers because of that reason. In addition, it is very difficult to write large projects with such a language, as there are no built in mechanisms to maintain consistency between different structures throughout the code and it is very easy to make mistakes that are almost impossible to detect in the development stages.

For our purposes, we can use Mozilla's SpiderMonkey, which is a simple JavaScript engine written in C++. It can be found under the ”latest-trunk/” in this link:

http://ftp.mozilla.org/pub/mozilla.org/firefox/nightly/

2.2 Basic language elements

2.2.1 Built in types

The language offers three basic data types:

- *boolean* — Holds a value of either *true* or *false*.
- *number* — Holds a 64 bit floating point. May also hold one of the special values *NaN* (Not a Number, the square root of a negative number for example) and *Infinity*. Note that the *NaN* propery is neither equel (==) nor identical (===) to itself, so one must use the function *isNan* instead. Also note that there is no integer type.
- *string* — Holds an immutable sequence of zero or more Unicode characters. Literal *strings* are created using " or ' (for each literal, the quote charecters must match). Note that the escape charecter is /, and that *strings* have a *length* member for retrieving the number of charecters. Also note the there isn't a *charecter* type, and a *string* of *length* one is used instead.

2.2.2 Objects

An *object* is a referenceable container of name/value pairs. The names are identifiers or *strings*, and the values can be literals or expressions of any type, including other objects. Literal *objects* can be created with a notation of a set of comma-separated name/value pairs inside curly braces. For example:

myObject = {*name: "Jack B. Nimble", 'goto': 'Jail', grade: 'A', level: 3*};

Since JavaScript is loosely typed, the programmer doesn't use type names in declarations, and an object can bee seen (and used) as a hash table. The subscript notation is used to add, replace, or retrieve elements in the object. For example:

myObject['name'] will be evaluated as *"Jack B. Nimble"*.

myObject['secondName'] = *'Alice';* will add a new field to the object.

Alternatively , a possibly more convinient dot notation may also be used (note how an object propety may also be a function):

myObject.foo = *function()*{*return 0;*};

While objects can be implemented as hash tables, none of the hash table nature (such as hash functions) is visible.

The *for* statement has an enumeration capability over the properties of an object:

```
for (var n in myObject){
    if (myHashtable.hasOwnProperty(n)){
        // do something with myObject[n] ...
    }
}
```

Functions and arrays (to follow) are also implemented as objects.

2.2.3 Constructors

A constructor is also a useful way to define anything similar to the normal notion of a class. Like many other things in JavaScript, a constructor is a function. In a constructor we can define what are the fields and methods of any object that will be created using it. For example:

```
function person(age, name){
    this.age = age;
    this.name = name;
    this.toString = function() {return "Age = " + this.age + ", Name = " + this.name;};
}
```

Notice the use of *this*, it is associated with the returned object of the constructor (similar to Java or C++). More information about *this* can be found in part 3.3 of this document. Now, in order to create a new instance of person, we simply need to call the constructor with the *new* keyword before it:

```
var pers = new person(15, "Allan Turing");
pers.toString(); // returns "Age = 15, Name = Allan Turing"
```

2.2.4 Arrays

JavaScript *Arrays* are very similar to objects in that they can also be treated as hash tables, but with two important distinctions:

- Each *array* has a *length* member, which is 1 larger than the largest integer key in the array.
- The keys in the hash tables are only integers (where in an object they might be any identifier or *string*).

Continuing the non typed checked nature of the language, the values in the *array* are not typed, and a single array can hold mixed values of all types (*numbers*, *strings*, *booleans*, *objects*, *functions*, and even other *arrays*).

Arrays are declared using of the three following syntax rules:

```
var arr = [];
var arr = new Array();
var arr = new Array(n); // Create an array of length n, with undefined elements
```

Note that unlike some other languages, size decleration during the *array* decleration is not mandatory, and is only when wanting to initialize the *array* with a specific size (probably for performance reasons). When the programmer inserts a value into the *array*, the *array* grows automatically if needed, and the *length* member is updated accordingly. The following assignment demonstrates the notation for *arrays* literals (note how the *array* literal on the right is consisted of a *number*, a *string*, an empty *array* and an *object* with two *number* fields):

```
arr[99] = [1, 'a', [], {1, 2}];
```

The values in an *array* indexes that haven't been assigned to are *undefinded*, so in the above example the compiler might choose not to allocate memory to elements 0 through 98.

The combined behaviors presented above make *array* very well suited to sparse array applications and very convenient to use, but not well suited for applications in numerical analysis.

2.2.5 Functions

Functions in JavaScript are declared with the keyword *function* where the programmer would usually use the return type:

```
function foo(){
    // do something ...
}
```

While the programmer can define the input parameters, an actual call to the function can be made with any number of parameters. Excess parameters are ignored, and missing parameters are given the value *undefined*. In order to make writing functions with optional arguments even easier, the language offers access to an *arguments* array, which holds all the actual parameters sent by the function caller. A simple example would be to calculate the product of an unknown number of *numbers*:

```
function product(){
    var prod = 0;
    for (var i=0; i<arguments.length; i++){
        prod *= arguments[i];
    }
    return prod;
}
```

A call to this function can be made with any number of *number* arguments.

Parameter passing is done by value for the built in types. *Objects* are passed by refernce. *Functions* are the main way of creating different scopes, as simple code blocks between curly braces has no special effect.

Anonymous functions, or functions that are declared without a name, act as lambda expressions, and in general can be thought of as expression rather than statements. Anonymous functions are not nataurally recursive, since the programmer has no name to call the function by. For the same reason, this type of functions are usually being activated (called) right after decleration. However, one can perform recursion using the *callee* field of the function's arguments:

```
var factOfFive = function(n){
    if(n <= 1) return 1;
    else return n * arguments.callee(n-1);
}(5);
factOfFive; // returns 120
```

This can also be used for mutual recursion.

One good purpose for these functions is to simulate blocks. For example, say we would like to swap the contents of two objects with a single *number* field. Using a temporary variable in the main program scope (or even inside a curly braces scope that has no effect) will litter the environment. Instead, we can put the swap code inside an anonymous function like such:

```
var a = {x:7};
var b = {x:12};
function(p1,p2){
    var tempX = p1.x;
    p1.x = p2.x;
    p2.x = tempX;
}(a,b);
```

Notice how we call the function right after decleration (with a and b as the arguments), and that changes to a and b (function side effects) are kept after the function ends due to the fact that objects are passed by refernce. This way the temporary variable is completely local, and cannot be refered afer the block.

2.2.6 Prototypes

As mentioned before, the programmer may add new fields to *objects* using simple assignment. However, this action only affect the specific *object* on which we assign to. We can use the *prototype* field that is part of any *object* in order to add a new field to all variables of a certian type. This is very useful for both the logic of the program and in order to share the same code and data between different variables. The following example demonstrates how to add an area method to all variables of type Rectangle:

```
function Rectangle(w, h){
    this.width = w;
    this.height = h;
}
var r1 = new Rectangle(4, 3.1);
Rectangle.prototype.area = function() {return this.width * this.height ;}
var r2 = new Rectangle(0.1, 11);
var a1 = r1.area(); // 12.4
var a2 = r2.area(); // 1.1
```

Notice how both r1 (that was declared before the prototype action) and r2 (that was declared after the prototype action) recieve the new area method.

2.2.7 Built in functions

The language provides a set of basic functions. Some are specifically intended to use in web browsers, but others offer an easy way to manipulate *numbers*, *arrays* and *strings*, as well as some mathematic functions. A full list of these functions can be found under ”Built-In Useful Functions” in this link:

http://www.tutorialspoint.com/javascript/

2.3 Advanced language elements

2.3.1 Scopes

JavaScript uses lexical scoping. The variables scope is always of the method they are defined in. If they are not defined in the method, then they are of the global scope. For example:

```
var x = 5;
function f(){
    return x;
}
```

f() will return 5. However:

```
var x = 5;
function g(){
    var x = 6;
    return function(){
        return x;
    }
}
```

g()() will return 6.

Don't get confused with Lisp, when using a variable in a function, the variables do not refer to the variables with the same name in the calling environment, the resolution of references is done when the function is defined. For example:

```
var x = 6;
function f() {return x;}
```

(function() {var x = 7; return f();})() Will return 6 and not 7.

In addition, as far as scoping is concerned, blocks are completely ignored. Variables defined inside them are completely visible throughout the entire scope they are in (function or global).

2.3.2 Hoisting

Hoisting splits the var declaration into 2. For example, in:

```
var x = 5;
```

- The interpreter adds a new statement at the beginning of the function: *var x = undefined;*
- At the place of the original statement: *x = 5;*

In addition, there are no block scopes, the scope is of a function. These rules may cause unexpected behaviour. For example:

```
function f(matrix, n){
    var x = 0;
    for(var i =0; i <n; i += 1)]{
        var row = matrix[i];
        for(vari = 0; i <row.length; i += 1){
            x += row[i];
        }
    }
}
```

If we call f with matrix = [[1],[2],[3]] and n = 3 we get an infinite loop, since i is always set to row.length at the end of each inner loop. To avoid this kind of mistake, you should always declare your variables at the top of the function!

Another example is the following code. If it was a C code, we would expect the code not to compile, as in the statement *return y*, the variable y is not declared. Thanks to hoisting, this code is fine in JavaScript and the function always returns *undefined*.

```
function g(){
    if (false)
        var y = 5;
    return y;
}
```

2.3.3 Functions and the *this* keyword

When we write a function in JavaScript, we can always use the *this* keyword. The *this* keyword contains a reference to the object of invocation. There are several ways to invoke a method in JavaScript, and in each way of invocation the *this* keyword has a different meaning:

- Method form — *obj.methodName(params)*. Calls the method and associates *this* with obj.

- Function form — *function(params)*. Calls the function, but *this* is set to the global object (explained later).
- Constructor form — *new function(params)*. *this* is associated with the new object that will be returned
- Apply form — *function.apply(thisObj, params)* or *function.call(thisObj, params)*. the caller defines what this is going to be.

The global object can be viewed as the object that all the global functions and variables are members of. When a function is not invoked with any object to bind *this* to, it will just point to the global object. The global object does not need to be expressed, we can just call its functions and variables directly. It does not need to be declared or initialized either.

Note that the semantic meaning of *this* in JavaScript is somewhat different than in Java. First, in Java, the *this* keyword can only be used in constructors and instance methods, and not in static methods where the *this* keyword would have no meaning (and is thus illegal). In JavaScript we can use *this* in all functions. In addition, in Java, inside a method, *this* will be associated to the object of invocation, where in JavaScript we do not have to specify an object of invocation, and in such a case the global object will be associated with *this*. Further more, in Java, inside an instance method, we do not have to use the *this* keyword explicitly, when not specifying it, the compiler treats a field/method x as this.x (unless x is a formal parameter or a local variable). In JavaScript we must specify the *this* keyword explicitly. Some examples:

```
// Java Code
class Person{
    private age = 0;
    public Person(){
        incrementAge();
    }
    private incrementAge(){
        age++;
    }
    public static staticIncrementAge(){
        age++; // Illegal, the method was not invoked on an instance
        this.age++; // Illegal, same reason (no this to refer to)
    }
}
```

```
// JavaScript Code
function Person(){
      this.age = 0;
      this.incrementAge = function (){
            //age += 1; // Will not refer to the object's age, but to a global var
            this.age += 1; // That's always ok!
      };
      this.incrementAge();
      incrementAge(); // Does not affect this
}
```

As we can see, the main difference is that in Java the *this* keyword gets its meaning from the context in which it is used, where in JavaScript it gets the meaning from the function call form.

2.3.4 Currying functions

Currying functions, named after Haskel Curry, is a method of transforming functions which expect 2 or more arguments and return a value, into functions that expect a single arguments, and return a function that expects the next arguments. More formally, Currying may transform a function *f: TxR->S* into *g:T->R->S*.

We can do that quite simply in JavaScript. For example, curried add will look like:

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

So, for example:

```
g = curriedAdd(5); //g is a function that expects another integer
g(3); // returns 8
```

2.3.5 Closures

As explained before, all variables (if declared with the keyword *var*) live inside the scope of a function. That, and the fact that we can return functions, implies that variables declared in functions can be used even after the function has returned, as we have seen before.

This implies that using a stack for the activation records might be problematic. The closures mechanism is used in order to solve this problem, and in essence can be implemented for JavaScript by the runtime environment managing its activation records as a graph on the heap, allowing multiple records for a single function, and using the garbage colletor to reclaim the memory for unreachable closures. The following code and image illustrates this behaviour:

```
function f(x){
    function g(y) { return x+y;};
    return g;
}

var h = f(3);
var k = f(4);
var z = h(5);
var w = k(7);
h = null;
```

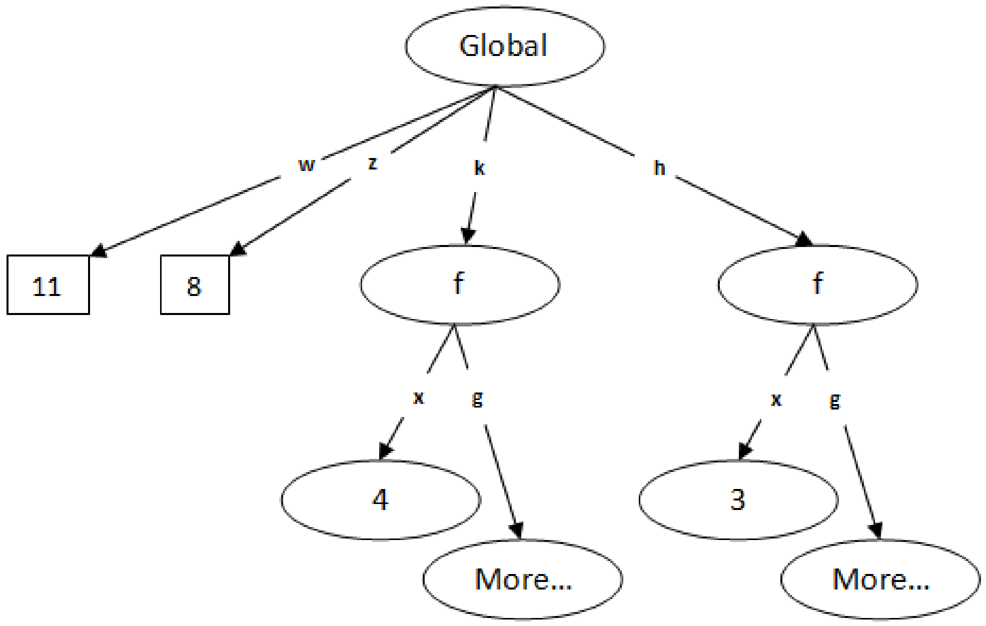

Figure 2.1: Closures implementation example

- Every time the function f(x) is called, a new activation record is formed, on the heap, and the calling environment (in this case, the global environment) points to that activation record.

- Since the returned function g uses the variable that lives inside of f (x), the activation record for f must not be deleted, so that when g is called (for example, when we invoke h(5) which points to g) x still lives and we can use it.

- Notice that since f(x) is invoked twice, we get two instances of the activation record of f, and for each call of g we get a different value of x.
- When h and j will die, the garbage collector will be able to release the activation record of all the fs, as they will no longer be needed.

A nice example for the benefits of closure is when using arrays and functions:

```
var arr = [1,2,3,4];
function f(n){returnarr[n];}
```

arr is global, and can be hidden by another global (if in another part of the script someone else used another variable called arr) which might be destructive. We can define arr inside f, but then we would create the entire array each time. Closures solves it:

```
function f(){
      vararr = [1,2,3,4];
      return function(n){return arr[n];};
}
```

Now arr lives once for each call of f(n). We may now call f, and save the result in a variable. Whenever we invoke the function returned by f, the same arr is used, but no one can hide it. For example:

```
var h = f();
h(2);
h(1);
```

In both cases, when invoking h, the same array is used.

Closures can also be very useful in maintaining object oriented principles. For example, encapsulation: When we set fields to an object, they are automatically public, and can be accessed and changed by the client. The solution can be obtained, once again, with closures:

```
//Constructor:
function ObjectWithSize(){
      var size = 0;
      this.incrementSize = function(){ size += 1;};
      this.getSize = function() { return size;};
}
```

```
// Now, lets call the function as a constructor.
// Notice that size is not defined in this,
// and we cannot access and change it from outside the constructor.
varobjectWithSize = new ObjectWithSize();
objectWithSize. incrementSize();
objectWithSize. incrementSize();
objectWithSize.getSize(); // Prints 2
objectWithSize.size = 8; // This is ok, but it just adds a new field to the object.
objectWithSize.getSize(); // Still prints 2!
```

2.3.6 Garbage collection

Like other scripting languages (and many other languages as well) JavaScript uses automatic garbage collection to release memory of strings, objects, arrays and, as explained in the closures section, functions. This sits well with one of the main design goals of the language which is to be as easy as possible. There are several ways an interpreter can implement JavaScript garbage collection, amongst them:

- Per page — releases the memory when the browser changes the page. This is probably the easiest to implement, and the most time efficient, but might be problematic as it might make the script very wasteful in memory. In the following scenario, we might get some very problematic memory leaks, as the browser does not change page throughout the function.

  ```
  function muchMemory(){
      for(var i = 0; i <veryLargeNumber; i += 1){
          var obj = new VeryLargeObject();
      }
  }
  ```

  In this example, the objects created (*VeryLargeObject*) are not released at all throughout the function and we have a definite memory leak. If *veryLargeNumber* and *VeryLargeObject* are large enough, this may cause some severe memory problems.

- Reference Counting — used in old implementations of JavaScript, such as Netscape 3. Every memory region has a counter, which counts how many times it is pointed by another variable. When another variable points to it, the counter increments, and when a pointer to it dies (for example, garbage collected) the counter decrements. When the counter reaches 0, the garbage collector can release the memory. This is a simple method with better performance. However, it has some difficulties in dealing with circular references, as can be seen in the following example:

```
function someObject(){
    this.attachOtherObject = function(otherObject){
        this.anotherObject = otherObject;
        otherObject.anotherObject = this;
    }
}
var firstObj = new someObject();
var secondObj = new someObject();
firstObj.attachOtherObject(secondObj); // Reference counting will not release both
firstObj = 5;
secondObj = 5; // Both objects still exist, even though no one references them
```

In this example, the objects created will never be released, as each of them always has a reference count larger than 0, even if they are not really reachable.

- Mark and Sweep — this is the most commonly used method of garbage collection today (in other languages as well). The garbage collector traverses all the variables in the environment (or in other words, the fields of the global object), and marks every memory region reachable from them. In the second stage, it goes through all the allocated memory and releases all unmarked memory. This method releases more unused memory, including circular references. Now, when we have a circular reference to an object (and no other references) it will still be released in the sweep stage, as the object will not be marked in the mark stage.

2.3.7 Inheritance

There are several ways to create a type that inherits from another type in JavaScript:

- Parasitic way — we create a constructor for the inheriting type, assign the parent type and do any changes we want to it. In other words, we create the original object, and then simply change it. For example:

```
function parent(){
    this.toString = function() {return ”in parent”;};
    this.bla = function() {return ”in parent”;};
}
var p = new parent();
p.toString(); //returns ”in parent”
function child(){
    var that = new parent();
    that.toString = function() {return ”in child”;};
    return that;
}
var c = new child();
c.toString(); // returns ”in child”
c.bla(); // returns ”in parent”
```

- Augmentation way — we simply create an object of the inherited type, and start adding/overriding methods and fields to it. It is quite similar to the parasitic way, only we actually create an object of the parent type, and only later add/change methods/fields. For example:

```
function parent(){
    this.toString = function() {return ”in parent”;};
    this.bla = function() {return ”in parent”;};
}
var p = new parent();
p.toString(); //returns ”in parent”
var c = new parent();
c.toString = function() {return ”in child”;};
c.toString(); // returns ”in child”
c.bla(); // returns ”in parent”
```

- The prototypal way — Crockford suggests another way to perform inheritance, not from a given constructor (like in parasitic and augmentation), but rather from an existing objects prototype. For example:

```
function parent(){
    this.toString = function() {return ”in parent”;};
    this.bla = function() {return ”in parent”;};
}
```

```
var p = new parent();
p.toString(); //returns ”in parent”
// This function creates a new object of the type of o
function object(o){
    function F() {}
    F.prototype = o;
    return new F();
}
var c = object(p);
c.toString = function() {return ”in child”;};
c.toString(); // returns ”in child”
c.bla(); // returns ”in parent”
```

We can see that in all three examples, the difference is not in the way we actually change/add fields and methods to the inheriting type, but how we actually obtain an object, on which we can perform the changes. Therefore, the fact that objects are referenceable containers of key/value pairs is the same here, after creating the object, we simply add keys and values or change the values for specific keys

2.3.8 Exceptions

An exception in JavaScript is like a C++ exception, in that it can be of any type (primitive, string, object), it can be thrown and it can be caught. Following is a simple try catch block, with string exceptions being thrown:

```
var x = prompt(”Enter a number between 5 and 10:”,””);
try{
    if(x >10){
        throw ”Err1”;
    }
    else if(x <5){
        throw ”Err2”;
    }
    else if(isNaN(x)){
        throw ”Err3”;
    }
}
```

```
catch(err){
    if(err == ”Err1”){
        document.write(”Error! The value is too high.”);
    }
    if(err == ”Err2”){
        document.write(”Error! The value is too low.”);
    }
    if(err == ”Err3”){
        document.write(”Error! The value is not a number.”);
    }
}
```

Note that in Java, for example, the type system is used for catching exceptions. When we catch an exception we must specify its type, and when an exception of that specific type is thrown it is caught by the catch block. That way we can use multiple catch blocks in Java, which can eliminate the use of the *if* used in JavaScript.

2.4 Further reading

- Brendan Eich, creator of JavaScript:

  http://en.wikipedia.org/wiki/Brendan_Eich/

- JavaScript and HTML:

  http://www.dummies.com/how-to/content/javascript-and-html.html

- JavaScript pros and cons:

  http://www.mediacollege.com/internet/javascript/pros-cons.html

- Simple JavaScript tutorials, including an online interpreter:

  http://www.w3schools.com/js/

- A complete documantation of JavaScript:

  https://developer.mozilla.org/en/core_javascript_1.5_guide

- Douglas Crockford's JavaScript portal, includes very nice explanations (including videos) about both simple and advanced features of the language:

  http://www.crockford.com/javascript/