Running head: JAVASCRIPT

JavaScript

Jeffrey W. Allen

University of North Alabama

Florence, Alabama

Abstract

This paper explores the JavaScript high-level, untyped, dynamic, interpreted programming language. First, a brief history of this this language is presented. This is then followed by in depth discussions of how the JavaScript programming language interfaces with computers. Throughout this paper, JavaScript may sometimes be referred to as JS.

JavaScript

History

Back in the early 1990's, there was an eruption of people that began using the web. It was almost as if the printing press was yet again recreated. Information was being shared amongst large numbers of people like never before. This was because of the introduction of new interfaces created specifically for these users to browse the web. The two main interfaces created were the graphical web browser, along with the Common Gateway Interface (CGI). The graphical web browser migrated web browsing done from the command line environment to a more intuitive environment which provided buttons and tools such as automatic URL lookups. With introduction of the Common Gateway Interface however, this allowed HTML (hypertext markup language) to request programs to be executed on the server.

This idea of clients continuously requesting information from the servers in order for something small to change on the client's computer became old though. A few companies such as Sun Cooperation, Microsoft, and NetScape, quickly came to the realize graphical web browsers lacked the ability to make computation associated with the rendered HTML. Sun's coopreration which owned the stable language Java, attempted to implement Java Applets that ended up being an overall failed design. NetScape, a company known for its own graphical web browser the NetScape Navigator, succeeded with decision of hiring a programmer named Brendan Eich to currate its solution for client side HTML computation, named Mocha, released in September of 1995. It was quickly thereafter renamed to LiveScript in September of 1995, and then renamed again to what we know now as JavaScript in December of that year. Microsoft saw the success JavaScript was having. So they used that design as a base for their own scripting language, named

JSscript.

With the fallout of Java Applets, and the rise of Microsoft's own flavor of a scripting language, Netscape quickly seeked to attain uniformity for its language by wanting solid language specification. Being that a language is an abstract idea, it is confusing for programmers to work together when no rules are specified. So This is where these coopretations offered the language up for standardization in 1996 to the European Computer Manufactuers Association (ECMA). Bickering ensued on what official name the standard should be released on, so in June of 1997, the language standard's name was given a default of "Standard ECMA-262 (ECMAScript core)". This lead to the Internation Standards Organization (ISO) to also recognize ECMAScript as an international standard ISO/IEC 16262. It is updated regularly, with this paper putting emphasis on the ECMAScript 5.1.

Design Goals

According to the ECMAScript 5.1 manual, ECMAScript (which emphasized as being simply a standardization reference to JS) was originally designed to be a web scripting language. A mechanism to which animates web pages in browsers to perform computation as a part of a Web-based client-server architecture. This follows closely to what Brendan Eich in 2008, a lead developer of JavaScript who now works for Mozilla, explained. This is when he was asked the question "What was your main goal in developing JavaScript?" To where he replied, "The idea was to make something that Web designers, people who may or may not have much programming training, could use to add a little bit of animation or a little bit of smarts to their Web forms and their Web pages." With this overall goal in mind, it inherently created issues related to design trade-offs. As will be seen throughout this paper, many decisions in the design were in favor of allowing a programmer to execute programs quickly. This comes at the expense of a program reliable.

In the battle between reliability and execution efficiency, it is apparent the latter won. One of the main reasons being Eich chose the language implementation of interpretation. This is where the source code is interpreted by a program which and executed directly on the machine. No translation occurs and object code is not produced for the program at all. The choice of the language being purely interpreted was driven by of one of the reasons Java Applets failed. Which was the heavy nature its intermediate code generator.

Java Applets took a hybrid approach of complitation and interpretation by translating its high-level language programs to intermediate language machine code to run its programs. Any compilation on the native machine at all made it slow and difficult for users to just browse web pages. Users simply did not want to wait for a program to be translated to native machine code, but rather quickly begin its execution. This decision also allowed JS to have even greater platform independance. The interpreter is usually embedded into popular graphical web browsers such as Google's V8 interpreter inside Google Chrome's browser, and Mozilla's Spidermonkey inside of Firefox.

The draw-backs of this though is that it is 10 to 100 times slower than a program which is compiled. Slowness is due to complexity of decoding high-level statements. To which each time the interpreted JS program is ran, it must be decoded every single time. Another disadvantage includes the fact the source program requires more space. Although Google's V8 engine attempts to resolve this by making multiple passes through a source program to reduce redundancy of defining statements.

Syntax

Syntax describes the actual form of the language's expressions, statements, and program units. The standardized structure of statements can be found in the language's specification, ECMAScript. In a 2008 interview, Brendan Eich explains that ”we were pushing it (JavaScript) as a little brother to Java, as a complementary language like Visual Basic was to C++ in Microsofts language families at the time. And it took off. We got it out in time." This is one of the main reasons JavaScript's own case-sensitive syntax heavily resembles that of Java and C.

A language's syntax can be combined in different ways in order to form different semantics. A language's semantics describe actual meaning of these syntactic constructs. ECMAScript 5.1 utilizes the EBNF (Extended Backus-Naur Form) metalanguage in order to describe its own syntax and semantics. Productions are represented as text residing on the left-hand side (LHS) of the the assignment statement, represented with two colons, followed by a list of references to mixture of tokens, lexemes, and references to other abstractions. Douglas Crockford, a noteable individual that is quite active the JavaScript community created syntax diagrams which graphically represents these productions. This is included in Appendix *A*.

As stated before, this language is interpreted. Google has developed its own interpreter, the V8 Engine, which is an open-source project written in C. V8 analyzes syntax by creating its very own abstract syntax tree from the source code with a file aptly named "AST.h", which stands for Abstract Syntax Tree. This checks to see if the code written is in fact valid JavaScript. The example below contains the necessary code to check if a JS `for` statement is valid.

```
class ForStatement FINAL : public IterationStatement {
 public:
  DECLARE_NODE_TYPE(ForStatement)

  void Initialize(Statement* init,
                  Expression* cond,
                  Statement* next,
                  Statement* body) {
```

```
    IterationStatement::Initialize(body);
    init_ = init;
    cond_ = cond;
    next_ = next;
  }

  Statement* init() const { return init_; }
  Expression* cond() const { return cond_; }
  Statement* next() const { return next_; }

  bool may_have_function_literal() const {
    return may_have_function_literal_;
  }
  void set_may_have_function_literal(bool value) {
    may_have_function_literal_ = value;
  }

  virtual BailoutId ContinueId() const OVERRIDE { return continue_id_; }
  virtual BailoutId StackCheckId() const OVERRIDE { return body_id_; }
  BailoutId BodyId() const { return body_id_; }

  bool is_fast_smi_loop() { return loop_variable_ != NULL; }
  Variable* loop_variable() { return loop_variable_; }
  void set_loop_variable(Variable* var) { loop_variable_ = var; }

 protected:
  ForStatement(Zone* zone, ZoneList<const AstRawString*>* labels, int pos,
               IdGen* id_gen)
      : IterationStatement(zone, labels, pos, id_gen),
        init_(NULL),
        cond_(NULL),
        next_(NULL),
        may_have_function_literal_(true),
        loop_variable_(NULL),
        continue_id_(id_gen->GetNextId()),
        body_id_(id_gen->GetNextId()) {}

 private:
  Statement* init_;
  Expression* cond_;
  Statement* next_;

  // True if there is a function literal subexpression in the condition.
  bool may_have_function_literal_;
  Variable* loop_variable_;

  const BailoutId continue_id_;
  const BailoutId body_id_;
};
```

Expressions and Assignment Statements

According to Sebesta (2012), "Expressions are the fundamental means of specifying computations in a programming language." Simply put, an expression in JS is a phrase which the interpreter can evalute. It is further explained that "The precondition and postcondition of an assignment statement together define precisely its meaning" (p. 150). These two concepts provide the building blocks for how results are produced by programs and algorithms. In this section, it will be explained how these fundamental concepts of computing are accomplished by JavaScript.

A paradigm in programming describes the style of structure and elements of computer programs. JavaScript itself utilizes multiple paradigms. It can be either object-oriented, imperative, and even functional. Object-oriented, which is usually paired with the imperative paradigms, allow for the state of a program to change based on the sequential assignment of objects. These objects can consist of abstract data types which can inherit from other objects to which attributes can be dynamically bound. The lesser used functional paradigm allows JS programs to produce results without the use of assignment statements of variables. Programs instead are controlled soley by function applications and conditional expressions. An example of this can be seen the way JavaScript's functions are treated first class objects.

This will produce a value. The syntax diagram below describes an expression.

When an assignment occurs in JavaScript, both a variables type and value is bound. This is just the nature of dynamic typing. JavaScript does follow the current trend of using a single equal sign for assignment. It also supports compound assignment operators, conditional targets, unary assignment as well. For example:

```
var number = 5;
number = number + 1;
document.writeln(number); // Writes 6 to HTML DOM
number += 1;              // Compound operator for addition
document.writeln(number); // Writes 7 to HTML DOM
```

All the operators from Java are included in JS. There also includes two new equality operators, === and !== operators. These check to see if the two operands on either side of have the same type and same value. According to Crockford, these new operators do not coerce values of the operands being checked if they are of different types, the way the old == and != operators do.

```
if ('' == '0')  // false
if (0 == '')    // true
if (0 == '0')   // true
if (false == 'false')     // false
if (false == '0')         // true
if (false == undefined)   // false
if (false == null)        // false
if (null == undefined)    // true
if ('\t\r\n' == 0)        // true
```

To understand the evaluation of an expression in JavaScript, it must first be understood the precedence in which operators are assigned. The operarators which have the highest precedence over all others are the refinement and invocation operators. These are things like the `.`, `[]`, and `()` operators. The next set of operators evaluated by JavaScript interpreter are unary operators. These include operators such as `delete`, `new`, `typeof`, `+`, `-` and `!`. Multiplication, division, and remainder operatations are all viewed by the interpeter as having the same precedence. The next level of precedence below these operators include addition, concatentation and subtraction operations. These are followed by the inequality operators, such as >=, <=, >, and <. Equality operators, such as the ones in the example above, are evaluated next. The logical operations are evaluated next. Where the logical and operation has precedence over the logical or. Where lastly, the ternary operator is evaluated. It is important to note that JS does not support overloading operators.

Data Types

JavaScript has a set of primitive data types and one complex data type. The primitive data types are not defined in terms of other types. These include `number`, `string`, `boolean`, `undefined`, and `null`. The complex data type `object` is a collection of properties, where each property is a name-value pair.

The only numeric type in this language is `number`. All numbers in JavaScript are represented in memory as IEEE-754 floating-point format. All integers between -2^{53} to 2^{53} can be represented in memory exactly. Floating-point literals, or real numbers, however causes problems when relation and arithmetic operation occurs. This is because values like 0.1 cannot be represented in memory precisely. The following example shows how this can cause problems.

```
// var x = .3 - .2; // thirty cents minus 20 cents
// var y = .2 - .1; // twenty cents minus 10 cents
// x == y           // declared false
// x == .1          // also declared false
// y == .1          // surprisingly declared true
```

The `string` type allows for JS to represent text. It uses a 16-bit unsigned value used to represent a single unit of text. The UTF-16 ecoding of the Unicode character set is used to decipher which character is which. For example, the character "a" in JavaScript is actually the hexidecimal number 0x0061. Variables can be assigned literal values of strings by enclosing characters inside of quotes. These can be single, or double quoted. Explained later in this chapter, relational operations are made easier, such as pattern matching, with the the regular expression object.

Next, the `boolean` data type. It is the simplest data type in JS. This is because they can only be either one of two values, true or false. This is usually represented in memory as a single byte with either a value of 0 or value of 1.

The last two primitivie data types `undefined` and `null` are special. These two keywords in the language evalute to indicate there is no value. However, a curious event occurs when the `typeof` operator, an operator which returns the parameter's type as a string value, is used on `null`. The type object is returned. So in reality, it is a special type of object, but in real-world applications it is simply a method for explaining there is no value.

The last data type explained in this section is `object`. Rather than a single value being associated with this type, an object is a compound value. It is an unordered collection of properties with eaching having a name, and value associated with that name. A more common name for this data structure an associative array, or dictionary.

This is where JavaScript greatly showcases its object-oriented abilities. Abstraction, encapsulation, inheritence and polymorphism are all attained through use of this data type. It must be carefully observed the way inheritence is executed in this language though. The object which is inherited from is named a "prototype". Unlike in Java and C++ though, once an object inherits from its prototype, it is viewed as its own entity. This means when an object inherits all properties from its prototype, and the properties of the prototype is modiefied in any way (deleted, added, ect.), it does not affect the child object.

Arrays are a special type of object. An array is untyped, meaning that each element may be of a different type. The elements in this object are ordered, beginning at the 0 index. Theses arrays described as being heap-dynamic. This means the binding of subscript ranges and storage allocation is dynamic. They have the ability to change any number of times during the lifetime of the array. When attemptting to access and index which does not exist in an array, `undefined` is returned. This is common in sparse arrays, where there exist gaps between the indices of elements.

A wide variety of operations can be executed on an array. JavaScript 1.8 supports: addition of elements, addition of other arrays, deletion of elements, slicing or splicing, reversing, sorting and modification of elements can be done. There is also an operation which returns the length of the array, suitably named `length`. It is later explained how arrays can be used to achieve stacks and queues.

Data Structures

JSON.

ECMAScript 6 will introduce classes.

Conditionals

Conditional statements allow for the flow of a program to differ. As previously stated by Brendan Eich, JavaScript closely resembles Java. So with Java being an imperative language, computation is accomplished by assigning results of evaluted expressions to variables. Sometimes though, evaluations of statements need be alternatively chosen based on certain conditions. This concept gives programs power and flexibility by allowing different paths to be chosen and allowing for repitition of statements. In order for a program to decide the pathway and number of repitition iterations, logical control statements, counter controlled statements, or a combonation of the two are used.

Path selection of a program is based on selection statements. These provide the means of allowing to choose between two or more execution paths of a program. There exist two-way selection statements, along with multiple-selection statements. The two-way selection statements in JavaScript include `if`, `if-else`, `if-elif`, and `switch`. All of these statements change the flow of the program based on whether an expression evalutes `true` or `false`.

Flow control statements which provides repetition functionality to a program are called iterative statements. These can be contrived of a single statement, or or a collection of statements to be executed 0, one, or more times. Today, they are mostly referred to as loops. These include the `while` loop, and `do-while`, `for` loop. The only difference between the `while` and `do-while` loops is the fact `while` is entry controlled opposed to exit controlled like `do-while`.

According to Douglas Crockford, `false` values of expressions are defined. To which all other values are true. False values include the literals `false`, `null`, and `undefined`. Along with those values, the empty string `''`, the number `0`, and the number `NaN` also declare an expression to be false. With all other values being true. Even the string '`false`'.

Subprograms

Subprograms in JavaScript are only known as functions. Even though some subprograms should be called procedures, in that some are simply a collection of statements that define parameterized computations, subprograms are named functions in JS. These special objects are able to be defined only once in the program, and have the ability to be invoked any number of times. JavaScript's function headers begins with the reserved word `function`, followed by an optional name, and an invocation expression. The invocation expression is made up of two parenthesis `()` that contain optional parameters, and a curly braces that wrap around statements, which make up the function's body.

A definition may include a list of identifiers, called parameters, that are used as local variables throughout the function's lifetime. When too many arguments are passed into a function's parameter list, they are ignored. Conversely, when there are not enough arguments passed into a function, they remaining parameters are assigned the value `undefined` in the function.

An interesting note about functions in JavaScript is that they are automatically brought to the top of the scope in which they are defined in. *Hoisting* is the official term of this action upon functions. Hoisting allows functions to be called upon before they are defined in the scope they exist in. An example can be seen here:

```
hoistedFunction();   // Successful call without previous definition
```

```
function hoistedFunction()
{
    document.writeln("Test Hoist");
}
```

Functions in this language can be either a statement or an expression. This is because they are treated as objects, known commonly as first class functions. This fact entails the ability for a whole function assigned the the value of another object. When an object contains a function, it is known as that object's method. When a function is invoked on, or even through an object, that object is a assigned the special property of being the function's invokation context. Therefore, the object is known in the function's environment as the value `this`.

The definition of a variable in a function is referred to as its referencing environment. Local variables, such as `this`, can either be defined statically or dynamically. JavaScript uses stack dynamic local variables. The main advantage of this decision is that storage is bound a variable when the program begins and unbound when the function finishes. This allows functions to be nested within each other, opening up the opportunity for recursion.

JavaScript's decision of defining variables as stack-dynamic, like with many other decisions that must be made when designing a programming language, created trade-offs. The overhead of memory allocation, initialization and deallocation when stack-dynamic variables is one trade-off. Another trade-off based on this decision is that variables must always be referenced indirectly. In the next section "Parameter Passing", it is seen how JavaScript's variables directly affected by this decision when it also observes parameters that are passed-by-sharing.

An explanation of closures should help clear up any confusion in understanding referencing environments. A closure is the term for a subprogram and the referencing environment where it was initially defined. Excellent news related to this concept is that access to properties of an object can be limited to where only methods are able to access and mutate them. Rendering a property private. An example of this concept is demonstrated in the following code.

```
var candy = (function () {  // Anonymous function 1

    var flavor = "Lime";

    return {
        changeFlavor: function (newFlavor) { // Anonymous function 2
            flavor = typeof newFlavor === 'string' ? newFlavor : "Lime";
        },
        getFlavor: function () { // Anonymous function 3
            return flavor;
        }

    };
}());

document.writeln(candy.getFlavor()); // Prints "Lime"
candy.setFlavor("Cherry");
candy.flavor = "Chocolate";
document.writeln(candy.getFlavor()); // Prints "Cherry"
```

The focus in this program is the `object` candy. There also must be emphasis drawn to the `()`, or invocation expression, on the last line. The invocation expression allows the function to return an object that contains two methods. Anonymous function 1 is the closure of the variable flavor, along with anonymous functions 2 and 3, `changeFlavor` and `getFlavor`. To conclude this example, it is seen with the print statements that the main program does not have access to the variable flavor. This is because it is out of the scope of the main program. Only the methods for that object have the privilege of accessing that variable.

Subprograms in JavaScript do not support the ability to be overloaded. If two functions are declared with the same name in both headers of the functions, invokation of the function that was defined last will be executed. Along with subprograms having the restriction that they are unable to be overloaded, operators in JavaScript are another entity which adhere to this rule. This constraint increases readability in the language by a minimizing the amount of referencing a programmer must do in order to undersand its newly adopted semantics.

A function in JavaScript is allowed to return a value of any type. The property of functions being first-class allow them to even return other functions. Yet, a functions have a restriction of only being able to return one value.

`eval()`

Parameter Passing

As explained before, a parameter of subprogram (referenced interchangabely as a function in this section) is an interface a calling program uses to allow subprograms to access their data. There are many details associated with the way a program written in any language program attains the data needed from an outside environment in order to execute. The details about how JS in particular attains access to the data it needs using parameters.such as methods a parameter are passed,, and even passing other subprograms as a parameter to another subprogram.

Local variables are declared

Of all the parameter-passing methods there are, JavaScript utilizes the ”pass-by-sharing”. Parameters are pass-by-value-result. Formal parameters are used. Positional parameters. Call-by-sharing.

It is obvious in this langauge that parameters are not checked for their types.

Data Abstractions

Abstraction is a view or representation of any entity that includes only the most signifcant attributes.

An instance of an abstract data type is called an object.

Concurrency

Ajax programming.

Recursion

Yes.

Exception Handling

JavaScript does exception handlers. These are special processing events that attempt to control the interference of normal program flow. The exception object

```
var try_example = function() {
  try {
    functionNotIsNotDefined("seven");
  }

  catch(e) {
    document.writeln(e.name + ": " + e.message);
  }
}
try_example();
```

Interesting enough though, JavaScript does not raise errors when overflow or underflow occurs. Only the values `infinity`, `-infinity`, or simply `NaN` (Not a Number) are printed out when events like division by 0 occurs.

Input/Output

Unusual Features

Contributions to the Programming Language Landscape

References

Personal Reflection

Consider the environment this programming language is being exposed to. Idiocy building on idiocy isnt something you want, and is very hard to decipher between intelligence when surrounded by bad. I hesitate to fall head over heels in love with this language.

People in the previous years had to wait a whole day for their code to compile. Waiting a whole day to find out there is a semi-colon missing provided incentive to pay attention to detail. JavaScript however is an interpreted language. This languages interpreter is being supported by a majority of browsers being used by people today.

Author Note

Patricia Roden