**A JavaScript Interpreter Implemented in Java**

by

Yao Shunyu, David

(1430003038)

A Final Year Project Thesis (COMP4004; 3 Credits)

submitted in partial fulfillment of the requirements

for the degree of

Bachelor of Science (Honours)

in

Computer Science and Technology

at

BNU-HKBU

UNITED INTERNATIONAL COLLEGE

December, 2017

DECLARATION

I hereby declare that all the work done in this Project is of my independent effort. I also certify that I have never submitted the idea and product of this Project for academic or employment credits.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

##### Yao Shunyu, David

##### (1430003038)

##### Date: \_\_\_December 14, 2017\_\_\_

**BNU-HKBU**

**United International College**

Computer Science and Technology Program

We hereby recommend that the Project submitted by Yao Shunyu, David entitled " A JavaScript Interpreter Implemented in Java " be accepted in partial fulfillment of the requirements for the degree of Bachelor (Honours) of Science in Computer Science and Technology Program.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

##### Date: \_\_\_December 14, 2017\_\_\_ Date: \_\_\_December 14, 2017\_\_\_\_

ACKNOWLEDGEMENT

I would like to express my great gratitude towards my supervisor, Dr. Philippe Meunier who had given me invaluable advice to this project, which I will appreciate for my whole life.

**A JavaScript Interpreter Implemented in Java**

by

Yao Shunyu, David

(1430003038)

A Final Year Project Thesis (COMP4004; 3 Credits)

submitted in partial fulfillment of the requirements

for the degree of

Bachelor of Science (Honours)

in

Computer Science and Technology

at

BNU-HKBU

UNITED INTERNATIONAL COLLEGE

December, 2017

# ABSTRACT

In the recent years of trend in software developing, the platforms people turn to build things on are transferred to more portable ones: web browsers on PC, or web browsers embedded in mobiles, even developing environment integrated in social network software (such as Wechat).

A compilation process is fairly long. Source code is parsed into an intermediate representation in the front end of a compiler by lexer, parser following by semantic analysis, and is optimized and translate to machine code to run in the back end in compilation, but interpreter needs less, it only need the front end of a compiler (sometimes not even a complete front end) and directly execute the intermediate representation. For compiled program, people need to have the proper runtime environment, the proper configuration and installation in local, while by interpretation, you only need to install the platform for all the software applicable to it.

An interpreter of JavaScript implemented in Java is achieved in this project, with simplicity and flexibility. Based on a lexer and a parser generated by ANTLR4, a parser generator, and a visitor pattern for semantic analysis, the interpreter executes all the expressions, statements and part of the object system of JavaScript.

Contents

[ABSTRACT 6](#_Toc500860560)

[1. Related Work 8](#_Toc500860561)

[2. Motivation 16](#_Toc500860562)

[3. Assumptions & Tools 17](#_Toc500860563)

[4. Current Work 18](#_Toc500860564)

[4.1 Overall Design 18](#_Toc500860565)

[4.2 Main Features 19](#_Toc500860566)

[4.3 Structure & Data Flow 22](#_Toc500860567)

[4.3.1 Input 22](#_Toc500860568)

[4.3.2 Lexer & Parser Analysis 22](#_Toc500860569)

[4.3.3 Semantic Analysis 24](#_Toc500860570)

[5. Conclusion 25](#_Toc500860571)

[References 26](#_Toc500860572)

# Introduction

An interpreter of JavaScript implemented in Java is achieved in this project, with simplicity and flexibility. Based on a lexer and a parser generated by ANTLR4, a parser generator, and a visitor pattern for semantic analysis, the interpreter executes all the expressions, statements and part of the object system of JavaScript, including calculations between integers, strings, Booleans, declaration and assignment of variables, function definitions and calls as both execution and constructors, and logic structures like If and Loop.

# Motivation

Among all the scripting languages, JavaScript ranks as the 2nd popular in 2017, and the most popular scripting language for web development. Thus, it seems actually quite favorable and pragmatic to develop an interpreter for JavaScript. It is flexible when it comes to the platform supporting it, deployable on both websites and devices, mostly on websites.

But why in Java? As what is mentioned in the literature review, most of the prevailing industrial “interpreters” are actually interpreter systems, with not only a conventional interpreter, but also a runtime compiler system ensuing to carry on the code analysis and optimize it by compiling it to the native code. This process, implicitly assuming that the environment is chasing for the highest efficiency for code execution, therefore the whole process focuses on the speed, leading it designed in C. This, of course, is under the condition that modern web browsers enjoy the performance boost that is just brought by excellence of hardware configuration on PC in recent years. But under limited contexts, which there are still plenty of them, speed is not the most pursued quality of an interpreter. On all kinds of devices, in the Internet of things, or even just merely for experiment on using the certain language to implement, language besides C can be justified enough to achieve the goal.

By implementation in Java, this project of JavaScript aims to achieve a prototype for JavaScript Interpreter with simplicity and flexibility, which only satisfies the basic requirements from ECMAScript, users can take it and modify it as their wish, and deploy them wherever they need. It is not specialized as what most of commercial interpreters in market are, a proper metaphor for this project is just like an ARDUINO in interpreter, having the making to be pure and simple.

# Related Work

1. About Interpreter

In computer science, an interpreter is a program that runs the input code written in interpreting language (or programming language) without completely compiling it, shortly to say, it executes program directly. Traditionally, a program written in high-level language is parsed and generated as an intermediate code, and then optimized and translated into machine code as a form called object code, which is relocatable and then linked by linker together with library instructions in object code, and finally becomes executable file in a full compilation sequence. However, the compilation process takes much time compiling before the program is ready to run, and the portability is strictly limited to processor, operating system and sometimes even the platform. Thus the interpreter is introduced. For interpreter written in low level language, it has the similar object files performs the same with what its high-level language describes, it’s just that it’s interpreted at runtime only when it needs to be run. For a high-level language written interpreter, it either performs the same with source code after parsing (Campell, 2005), or it takes intermediate code generated by other compiler and interprets it. (Kistler & Franz, 1999) For software development which has a shorter interpreting cycle than compilation cycle, interpreter would help developer modify their code constantly by saving time because there is no need for translation to native code, and the code optimization cycle can be shorter. It also increases the portability for a scripted program (such as JavaScript engine in various web browsers), and makes the programming more dynamic. However, the disadvantages are obvious, it runs slower than pre-compiled programs, and it needs to be interpreted every time it executes, while compiler translates the source code to native code which can be run directly by microprocessor independently.

What’s more, the boundaries between compiler and interpreter are not mutually exclusive, and the concept of interpreter are considerably wide, it can sometimes considered as a virtual machine, and even in a lower-level aspect, a microprocessor is an interpreter.

1. Java Virtual Machine

A Java Virtual Machine is a virtual machine that builds an environment on a computer internally and for executing Java program. The concept of JVM is to run Java bytecode without constraints to the platforms and implementers as long as they have a JVM, and provide a sandbox to prevent interference from host machine program, and as a matter of fact, a JVM itself is an interpreter. Apart from Java itself, there are many other languages that can be used to write a program that runs on JVM. To execute such a program written in a non-Java language, it is either interpreted by Java programs and run on JVM, or it is compiled to run on JVM as Java bytecode and even furtherly JIT compiled into assembly code for performance optimization. Therefore, distinguished interpreters or compilers are needed for different JVM languages. (Venners, 1996-2017)

A JVM is appended at the back end of our project to run the interpreter on, for our interpretation is implemented in Java, and that’s where a program that executes exactly the same as the source JavaScript code runs.

1. Just-In-Time Compiler

JIT Compiler is a compiler that does Just-In-Time Compilation, which, comparing to Ahead-Of-Time Compilation, compiles code to native at the runtime rather than in prior. For a complete compilation system, it has different compilation strategies, among which there is one called bytecode-based compilation (Stack Overflow Community, 2010). It basically means that the compilation system has an intermediate code stage implemented in bytecode (Croce, 2014). Traditionally, a compiler would take the source code to generate byte code at the front stage, and then the bytecode itself would then be directly interpreted, or executed in a virtual machine (exactly like what a JVM does) (Oracle). A JIT compiler takes the place of a virtual machine or a interpreter – it compiles the bytecode into native code by reading them partially (rarely fully, out of the consideration of optimizing speed, which is called lazy compilation) in runtime, and executes the native code. Both memory and speed overhead in the traditional static compilation and interpreter are reduced through JIT compilation: the AOT optimization is getting faster here because now the (JIT) compiler deals with bytecode, which is of lower level; a lot of runtime optimizations are now possible; the machine code execution is faster than interpreting bytecode. It’s just that, the startup delay becomes considerably big overhead now.

My project does not include the JIT compilation conception because of pursuing studying concept and design on interpreter, due to which I want to simplify the architecture.

1. About JavaScript

JavaScript is one of the core web programming languages, along with all the other two web development language HTML and CSS, featuring significant characters as high-level, untyped, dynamic and scripting language (Flanagan, 2011). It essentially has been the most supported language that does not need any plug-ins by most of the popular web browser.

The creator of JavaScript is Brendan Eich, who was recruited by Netscape Communications in 1995 with goal to enable Scheme programming language into its web browser Netscape Navigator. At the time Netscape was projecting a new language which could help web developers easily assemble web contents together. Just before Brendan was prepared to start, the company decided to establish cooperation with Sun Microsystems in order to cover its production of static programming language – Java into Netscape Navigator. Then the blueprint for the new language was then necessarily featured with similar syntax with Java. Eich then completed a prototype with such a conception within 10 days in May 1995. Originally it was named as Mocha during development, but after embedded in beta version of Netscape Navigator 2.0, it was renamed LiveScript, and officially switched to JavaScript in a later beta version till now (PR Newswire, 1995). Although Java and JavaScript have similar appearance both in syntax and name, but they are substantially two 2 different languages: JavaScript is prototype-based and dynamic, while Java is class-based and static.

JavaScript was handed to ECMA International by Netscape, by whom a standard specification for this language is desired so that other web browser developers could take this as a reference while embedding similar implementation in their own work. ECMAScript was published in June 1997 with the first version ECMA-262, and JavaScript became the most known implementation as a matter of course (ECMA International, 2016).

JavaScript is now widely supported in web browser industries and also implemented in game developing, network deploying mobile applications and so on. And of course, my Java implementation (interpreter) is for JavaScript.

1. JerryScript

JerryScript is a light-weight JavaScript engine specially designed for running ECMAScript language on computing-resource-constrained devices (less than 64KB RAM & less than 200KB ROM), mostly intended to Internet of Things devices, developed by Samsung.

It mainly contains two components: Parser, and VM (JavaScript Engine). Rather than a single parser, a Lexer, an Expression Parser, a Statement Parser, a function Parser and a Source Code Parser constitute the Parser. All the other Pasers are connected to the Statement Paser, and the Statement Parser interacts with Lexer. The Parser does not generate Abstract Syntax Tree, it produces bytecode instead, and like what the JIT Compiler process does in SpiderMonkey, the JavaScript Engine, or the interpreter, also takes the bytecode as the input. But what is different is that, it does only exist as a single interpreter, and due to the specialty of running on resource-limited devices, it is non-recursive so that it won’t accumulate tasks in stack. (Samsung OSG, 2015)

Jerry Script is a highly referable instance for my JavaScript interpreter, for its architecture and the designing concept is fairly clear and simple enough, it could be a perfect object in experiment.

1. Google Chrome V8

The Google JavaScript Developers have just been firing up a new component for Google Chrome V8 JavaScript engine: the Ignition interpreter. Again, this interpreter is used to tackle the initialization overhead problem caused by the internal baseline compiler and two optimization compilers in the V8 engine. (Google, Inc)

Originally, the JavaScript source code is directly put into the baseline compiler code-named Full-codegen and compiled into unoptimized machine code when running for the first time the engine sees the code, during which there is no byte codes generated in the middle of the process, meaning from source code straight down into the machine code. And after it gets hot, V8 gets it into two optimization compilers code-named Cranshaft and TurboFan ( McIlroy & Swirski , V8: Hooking up the Ignition to the Turbofan). Cranshaft is an older version of JIT compiler in V8, and TurboFan is just the new version with more supportings and faster speed. The call from baseline to optimization generates considerable overhead because the source must be reparsed in the parser for both Cranshaft and TurboFan, meaning that plus the one parsing in baseline, JS source can hereby be parsed for 2 to 3 times. About 30% of V8 heap is spent on JIT unoptimized code, and 33% of time is consumed in parsing and compiling. What’s more, the unoptimized code just can’t be simply chucked away because once the script is given more changes (such as type changing), de-optimization is needed, the code running should then be traced back in the baseline.

And then Google development group boosts the Ignition interpreter at the beginning stage, which takes the source and output the bytecode. Comparing to the machine code the compiler generates, bytecode generated by Ignition is fairly much more concise (50% to 25% size of equivalent baseline native code), thus we can reduce the use of memory, especially by cutting memory occupied by functions that do not run very often; Bytecode lowers the level of the source from abstract syntax tree to bytecode, reducing time spent on parsing (only one time of parsing is needed), and also contributing to simplification the optimization/de-optimization structure complexity. (McIlroy & Hodson, Ignition: An Interpreter for V8)

As for the pipeline, the V8 engine after internally enabling ignition interpreter takes the source JS code and then interprets it; it would send it to baseline compiler and execute it after compilation; and even further, it sends bytecode into the optimization compilers if code gets hot. The baseline stage now has parser that parses the bytecode rather than source code anymore, therefore only one time does the source code need to be parsed; The optimized code can be directly de-optimized back to bytecode. It’s just that now the parser for both baseline and optimization take bytecode as input instead. (McIlroy & Hodson, Ignition: V8 Interpreter)

The JavaScript interpreter system in Google Chrome V8 is actually an interpreter following by multiple JIT compilers, which presents in a very mature commercial interpreter formation and my project does not bother including it since its complexity is unnecessary.

1. SpiderMonkey

SpiderMonkey usually refers to the JavaScript compilation process as a whole, or, a so-called JavaScript engine, embedded in the Mozilla Firefox web browser, which, regarded from the traditional definition of interpretation, forms a complete code compilation and optimization system instead of a single interpreter.

The beginning of SpiderMonkey is a pre-compiler (preprocessor) taking JS source code as input and outputing the script containing bytecode. The generated bytecode is then sent to the SpiderMonkey interpreter. Since this section of compilation, the main optimization of SpiderMonkey on code processing speed takes place. In SpiderMonkey interpreter, code processing becomes pretty slow as it only goes until Abstract Syntax Tree (AST) generation. But rather than stops there, it gathers type information of a certain piece of code, and records its hotness (the frequency of the code block). While a block of code turns to be of a certain level of hotness, the interpreter would then stop interpret it and throw it to JIT compilers. Right here, the JIT compilation process is not simply composed by one JIT compiler, but two separated, leveled ones. The hot bytecode would then go to the first JIT compiler, generating native machine code to run on microprocessor, as a warming-up, based on which the hot bytecode can also be monitored with more information gathered. If the code gets really hot, then it would finally go to the second tier JIT compiler code-named IonMonkey. IonMonkey does the same thing with the warm-up JIT compiler, it’s just the compilation speed and the code of different hotness they deal with are distinguished. (Mozilla Developer Network, 2016)

The original warm-up JIT compiler is JägerMonkey, and there was still considerable speed redundancy even through two leveled JIT compilers. The bytecode generated by JägerMonkey is highly related with type information, while both JägerMonkey and SpiderMonkey do not collect type information, which may constantly make bytecode invalid; and the calling process from JägerMonkey to IonMonkey can also be very time consuming. (Mozilla Developer Network, 2014) TheMozilla SpiderMonkey development group then replace the old JägerMonkey in Firefox with a baseline compiler that does not have a code name like “XXXMonkey” in 2013. The baseline compiler is of somehow a frame between SpiderMonkey interpreter and IonMonkey, considerably decreasing the speed redundancy by using inline cache chains and feeding more information to IonMonkey. (Vijayan, 2013)

Since SpiderMonkey and Chrome V8 are competitors to each other, it has a similar architecture with Chrome V8, which is again a purely chase for speed. My project does not include such a structure with JIT compilation system, aiming to keep the scale clear and simple.

1. Rhino

Rhino is an open-source JavaScript engine implemented in Java and nanaged by Mozilla, and it is also one of the only two active JavaScript engines project written in Java while most of the JS engines are written in C/C++, the other one is Nashorn, which is developed by Oracle and has been currently open-sourced since 2012. It essentially allows both javaScript and Java scripting. Rhino contains two internal components: a Rhino JavaScript shell which runs JavaScript code, and a JavaScript compiler which translates JavaScript source code to Java class files which are then awaiting for being loaded and executed. It basically appears as a java implementation of JavaScript, enabled on JVM. (Mozilla Developer Network, 2014)

Rhino is taken as another one of the most significant references in architecture design due to its implementation in Java, the implementation of embedding JVM to complete the interpreter is the part that influences the back-end of my project a lot.

1. ANTLR 4

ANTLR is a series of parser generator, with the newest version of 4. Regardless the definition of “Parser Generator”, it generates components of a compilation process for not only parser, but also lexer, and part of semantic analysis which traverses the generated abstract syntax tree. It is designed to deal with LL(\*) input grammar for parsing. ANTLR is currently managed under professor Terence Parr of University of San Francisco, and ANTLR 4 is now only targeted to Java, JavaScript, C#, Python 2, Python 3 Swift, C++ and Go. (Parr, 2012)

In my Final Year Project, I use ANTLR version 4 to help me generate the corresponding lexer, parser and the tree visitor template (in order to assist in implementing semantic analysis) for my JavaScript interpreter.

1. Abstract Syntax Tree

An abstract Syntax Tree is a sort of data structure that represents the structure and the content of given code. It is usually the output of a parser from compilation front end. For a compiler, AST is the intermediate representation of code that will be translated to assembly code in the back end, for an interpreter, it is directly executed.

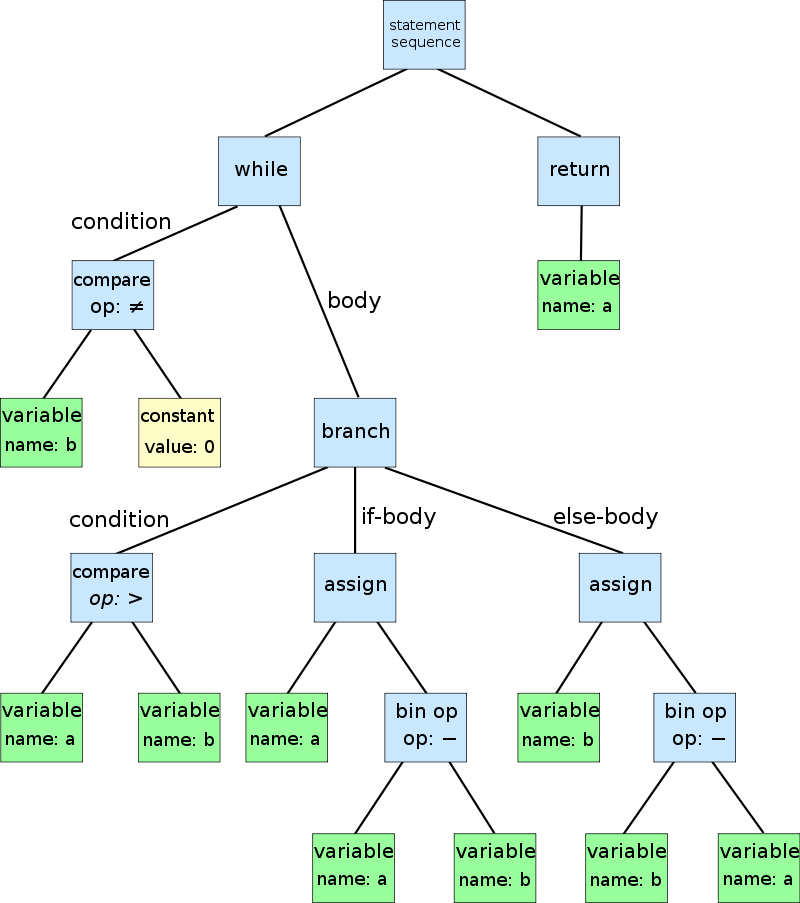


Figure : An abstract syntax tree for the following code for the Euclidean algorithm

while b ≠ 0

if a > b

a := a − b

else

b := b − a

return a

# Assumptions & Tools

Just because simplicity is what this project is supposed to be, the whole environment of this project only requires a Java Compiler and a JVM, which means users only need to download and install the official Oracle Java package.

Besides Java environment, ANTLR4, which is a parser generator, is what this project exploited during the implementation in order to assist the generation of lexer, parser and the tree visitor template.

Because of the partial development to get the current work on stage, this interpreter is now only accessible by command line interface by running in JVM.

# Current Work

The development of my interpreter project has been finished which includes: Grammar files to specify lexer & parser grammars, lexer and parser Auto-generated by ANTLR4 through these grammars, and two parse tree visitors which enforce the semantic analysis.

The lexer and the parser are automatically generated based on the definition of both the lexical grammar and parsing grammar written in a .g4 file, which uses regular expressions and context-free grammars. Then together with lexer and parser, there is an interface of parse tree visitor generated for further implementation. The definition for the semantic rules which tells the visitor how to access the parse tree and how to compute values needed after tree traversal is then implemented through .java files as visitors, which contains all the methods that separately visit each node in the parse tree.

## Overall Design

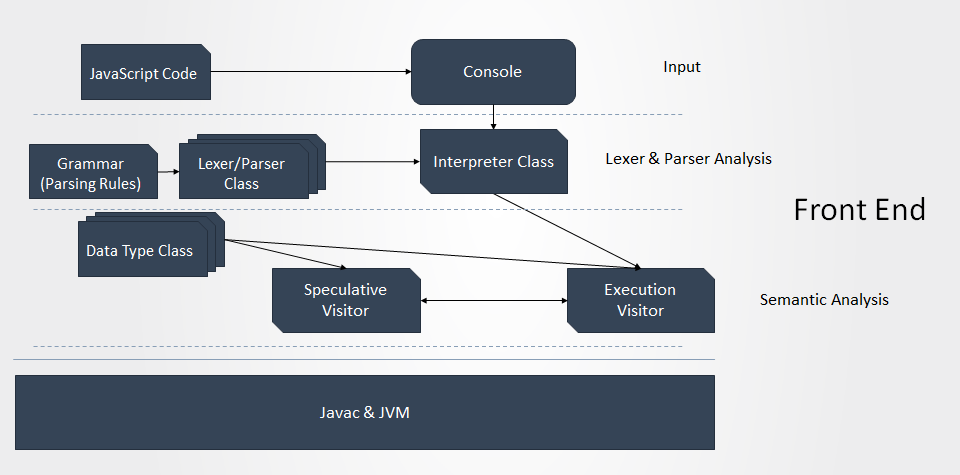


Figure : Overall Design

The overall design of my current project is shown in figure 2. Like a conventional interpreter, it has a lexer to do lexer analysis, a parser to do syntax analysis, and no back end to generate local codes, JVM directly runs the code’s meaning instead. But for the semantic analysis stage, I choose visitors to visit Abstract Syntax Trees.

## Main Features

Before going through the details and methodology of the whole system, several significant features should be explained in order to better understand the design.

* + 1. **Type System of JavaScript**

JavaScript is a weak (dynamic) typed language, which means that is does not have type assigning to the variables before them while declaration. Only tags of “var” can be put in front of variables, but even this tag is dispensable and makes no difference for some situations. This can be demonstrated by the codes below:

var a = 1;

b = 2;

But JavaScript needs to do the type checking anyway in order to do the correct value assigning and calculation, this is when we need the concept of Tags. Tags are like types, but instead of attributes of variables, they are attributes of values assigned to the variables. It is an inherent property of values, for instance, the number 1 is naturally being a “number” without the need to say “1 is a number”. Here in Java implementation, the data type classes themselves are tags for values. With all above mentioned, variables are just names for values, they can be numbers at first, and then strings later, hence “Dynamic”.

* + 1. **Visitor Design Pattern**

In object-oriented programming, a visitor design pattern deals with a static structure, but constantly changing tasks need to be done with the same structure. A visitor would traverse all the elements in the structure and complete a certain task, while another visitor could traverse the same structure doing a completely different task.

Here in my project, since JavaScript needs to traverse the whole code twice, with first time Speculative Parsing, and second time execution, visitor pattern will work perfectly. Each lexeme in the AST is represented as a method in each visitor, completing corresponding tasks.

* + 1. **Speculative Parsing**

This parsing stage is specialized for JavaScript in the Semantic Analysis stage. Before one going-through (parsing) of code for execution, there is another time of going-through to speculatively parse the code, and what it does is just basically put all the “declared” variables with tags “var”, “function”, “this.a”, ”obj.b” in to the current scope of codes without assigning any value. All these variables will already be there before execution, except that they are all “undefined”.

var a = 1;

function foo(){

alert(a);//undefined

var a = 2;

alert(a);//2

}

foo();

Like the code shown above, according to our thinking of conventional scope system, variable “a” in the function “foo” can not be used before it is declared, therefore the first “alert” function must crush. However, in JavaScript, “a” is still able to be used (even though it’s not defined). It is because speculative parsing put it in scope first, despite without value, before the execution.

What speculative parsing does is just basically for the enforcement of JS’s scope system. Because ultimately, the author of JavaScript wanted it to be scope-free. Later on there were demands for local variables, thus speculative parsing is added on to implement the concept of local variables.

This mechanism is perfectly implementable by Visitors. Visitor will visit each element once in the system once in order, instead of visiting each unit separately. Here in interpreter, visitors visit each node (method specifying what to do with each semantic rules) in the ASTs in order. A speculative parsing visitor runs through the AST (storing the structure and content of the code) for once before execution visitor. Just because the variables with “var”, “function”, “this.XXX”, ”XXX.XXX” are put into sub scopes before the whole code is executed, they are “more visible” to current code running than upper scopes, therefore implementing local variables.

* + 1. **Scope System of JavaScript**

JavaScript only has a global scope, which means there are no more sub scopes in logic structures like If and For/While loops. However, a chain of scopes can still be established while recursively calling functions.

In the global scope, which is at the top of all scopes, variables declared with/without “var” have the same visibility.

However, when a new function is called, a new scope is attached to its upper scope, with its initial situation attaching to the global scope. When there are more functions called in this current executing function, more scopes are attached, and can only be attached to scope where the function being called is defined in order to keep a reasonable visibility. Variables with tags “var”, “function”, “this.a”, ”obj.b” are local variables due to Speculative Parsing.

1. var a = 1;
2. function foo(){
3. a = 2;
4. }
5. foo();
6. alert(a);//2
7. var a = 1;
8. function foo(){
9. var a = 2;
10. }
11. foo();
12. alert(a);//1

For the code above, for instance, comparing to the left, adding a tag “var” to variable “a” in the codes on right is going to make it a local variable, thus making the later “alert(a)” function alerts differently:

On the left, from line 1-4, variable “a” is declared with value 1, and then function “foo” is defined, in which “a” will be assigned with new value 2. Right now, there is only a global scope: [a: 1(integer); foo: {a = 2;}(function)]. Before executing function “foo”, it is speculatively parsed, with a new scope attached to the global scope, but no new variable is put into the scope because no variable in “foo” has a tag “var”. Now the scope chain is: [a: 1(integer); foo: {a = 2;}(function)] -> [(empty scope)]. When line 5 is executed, it traverses back to fine the function called “foo”, and execute the function. Just as what “foo” defined, the variable “a” needs to be assigned with new value 2. The interpreter traverses the chain to find variable “a” and assign it with 2. When 5 is finished, the attached scope is eliminated, and becomes: [a: 2(integer); foo: {a = 2;}(function)]. Thus, the alert at line 6 alerts 2 when “a” is asked.

On the right, from line 1-4, a variable in global scope is declared with 1, and a function that executes “var a = 2” is defined. The scope is [a: 2(integer); foo: {var a = 2;}(function)]. At line 5, a new function scope is attached to global scope, and speculative parsing sees the “var” tag of “foo”, thus creating a new variable “a” without any value and putting it into the function scope: [a: 1(integer); foo: {a = 2;}(function);] -> [a: undefined;]. When executing line 5, the interpreter traverses reversely looking for “a”, and it finds it at the function scope, thus assigning it with 2. Later on, every time interpreter sees “a”, it will always see the one in the function scope first and operate it, this “a” overrides “a”s in upper scopes, accomplishing the concept of local variables. Now the scope is [a: 1(integer); foo: {a = 2;}(function);] -> [a: 2(integer);]. After execution of “foo”, its scope is recycled, and the scope chain becomes: [a: 1(integer); foo: {a = 2;}(function);]. Thus the alert at 6 will still alert 1 because the global “a” has never changed.

* + 1. **Object System**

Everything in JavaScript is object. But because JavaScript is a prototype based language, its objects are without classes, all coming from one prototype of object.

But still, there are two types of objects: “Object” objects (or user defined objects) that can have variables and methods that are free to modify by programmers; “Number”/”String”/…… objects that are embedded to scopes as prototypes with methods and variables that are not free to define or override.

Objects can be instanced through functions, meaning functions are used as constructors in JavaScript while not used as a normal function call but with a “new”.

1. function foo(){
2. this.a = 1;
3. this.b = 2;
4. return 3;
5. }
6. var ret = foo();
7. var obj = new foo();
8. alert(ret);//3
9. alert(obj);//[object Object]
10. alert(obj.a);//1
11. alert(obj.b);//2

Just as the code shown above, function “foo” can be used as a constructor, but also a normal function call.

Line 1-5 defines “foo” as a function creating an object, assigning it with variable “a” with 1, and “b” with 2 by line 2-3, and returning value 3. At line 6, “foo” is used as a function call, thus even though when “foo” is executed, and there is an object with “a=1, b=2” created, only the return result of function “foo” is assigned and kept in variable “ret”. The return of “foo” is 3, thus alert of “ret” at line 8 shows 3; At line 7, “foo” is used as a constructor, which means its return operation does not matter anymore, the object it creates is kept by the variable “obj”. Because “foo” is a user defined constructor, meaning “obj” is a user defined object, thus line 9 alerts a “[object Object]” indicating it is user defined; and because the “obj” object is initialized by “foo” with “a” and “b” in it, thus their values are shown in line 10-11.

## Structure & Data Flow

## Input

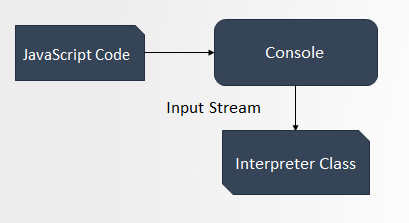


Figure : Input

In the input stage, console takes input JavaScript code and transforms it into input stream, passing it to interpreter. Later on, the interpreter is going to transform the input stream to string and segregate them through lexer rules (Figure 3).

## Lexer & Parser Analysis

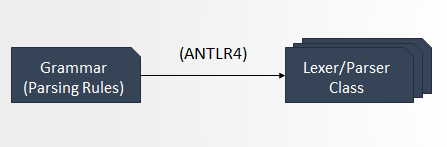


Figure : Lexer/Parser Generating

In the Lexer and Syntax (Parser) Analysis, lexer rules and grammar rules for JavaScript language are specified by the author in .g4 files. Antlr4 helps generate a lexer class and a parser class according the grammar .g4 files. Each rule is represented as a method in the class to specify what to do with the recognized lexemes (Figure 4).

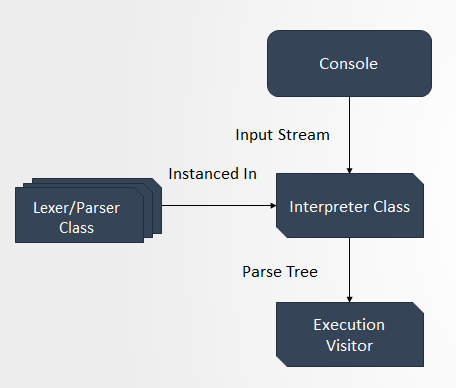


Figure : Abstract Syntax Tree Generating

Interpreter instances a lexer to process the input stream, segregating characters in the code to different types of lexens (for example, ‘a’, ‘b’, ‘c’ together without a space are recognized as an “id” for a variable; ‘=’ is recognized as an “equal mark”). Then parser is instanced to get these lexens as input and further recognize them as bigger groups of grammar elements (for example, an “id”, an “equal mark”, another “id”, and a “semicolon” in sequence are recognized as an “assignment”), but still not yet able to understand them. Interpreter class then outputs a parse tree (Abstract Syntax Tree) representing the structure and the content of given JS code, and pass it to visitors in next stage (Figure 5).

## Semantic Analysis

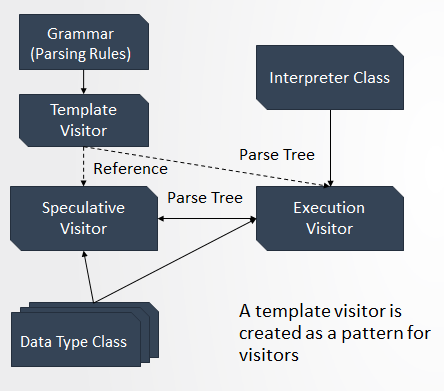


Figure : Semantic Analysis

In Semantic Analysis, Antlr4 also helps in generating a template for visitors, but nothing more than a template. It instructs me to specify what nodes (methods for grammar rules) can be in a parse tree derived from my grammar, and what exactly need to be done (what the method does) when each node is visited (calling the methods).

A speculative visitor class and an execution visitor class are defined. When execution starts, parse trees are passed to execution visitor instance, and inside execution visitor, when two situations met, speculative parsing visitors are instanced and sees through the parse trees fed: 1. At the beginning of all execution; 2. At the beginning of a function call. Then execution ensues for the code (parse trees) seen through.

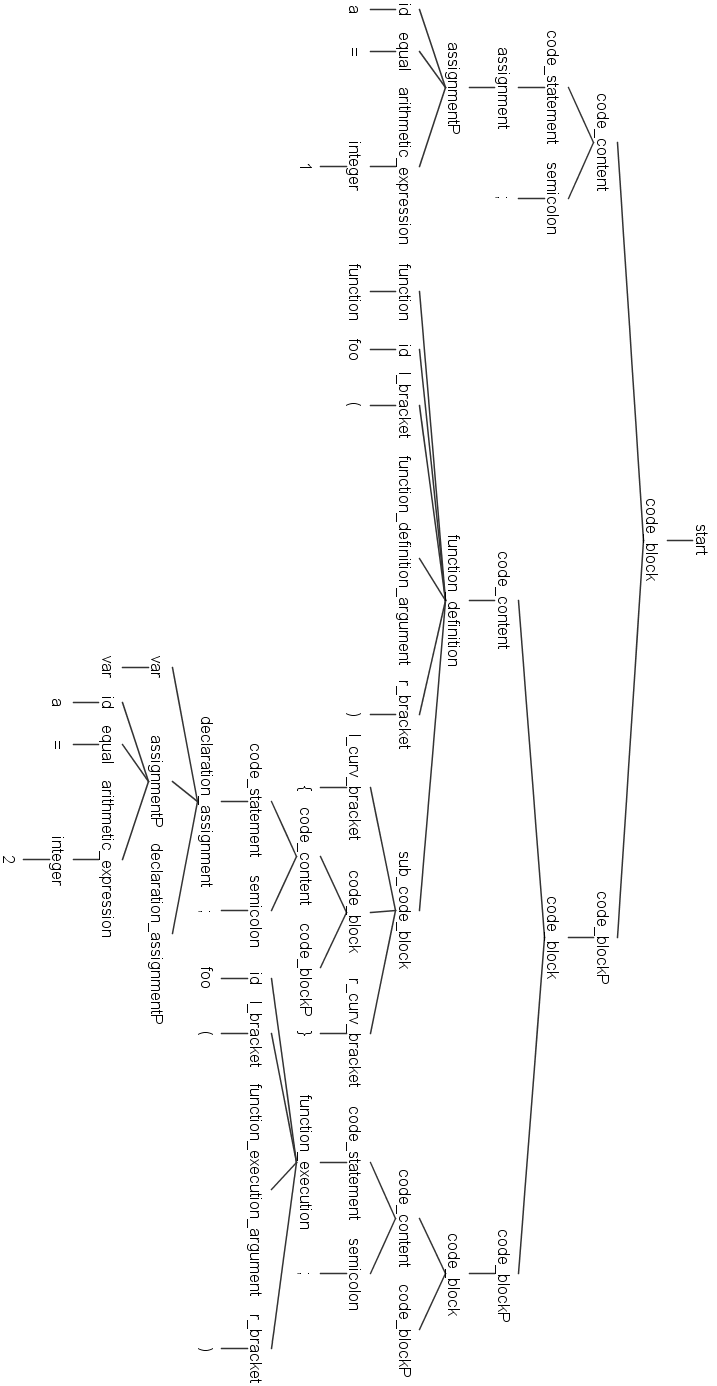
A code example can explain this clearer.

1. a = 1;
2. function foo(){
3. var a = 2;
4. }
5. foo();

Following page is a corresponding AST of code above.

Each node in the tree represents a corresponding semantic rule, an expression or statement in the code, matching a method completing semantic tasks (assigning values, calculations etc.).

While the code is executed first, the whole AST will go through speculative visitor for one time, and then start to go through execution visitor. While going through line 5, another speculative visitor starts to visit the AST that function “foo” represents, which is line 3. And then another execution visitor starts to visit AST of “foo” to execute it.



# Conclusion

Currently the interpreter can tackle with: the basic calculation on integers with addition, subtraction, multiplication and division, and calculation of basic appending strings to other strings; basic If and Loop structures; function defining & calling; variable declaration & assignment together with scope system; object creating and variable/method defining and using.

But there is still considerable work to be done in the future to complete the object system of JavaScript, including all the embedded objects not for user to define, also other features like overriding mechanism, etc.

# References

McIlroy, R., & Swirski , L. (n.d.). *V8: Hooking up the Ignition to the Turbofan.* Retrieved from docs.google.com: https://docs.google.com/presentation/d/1chhN90uB8yPaIhx\_h2M3lPyxPgdPmkADqSNAoXYQiVE/edit#slide=id.g1357e6d1a4\_0\_58

Campell, C. (2005). *AST Intermediate Representations.* Lambda the Ultimate forum.

Croce, L. (2014, 9 22). *Just in Time Compilation.* Retrieved from www.cs.columbia.edu: http://www.cs.columbia.edu/~aho/cs6998/Lectures/14-09-22\_Croce\_JIT.pdf

ECMA International. (2016). *Standard ECMA-262 ECMAScript® 2016 Language Specification*. Retrieved from http://www.ecma-international.org/: http://www.ecma-international.org/publications/standards/Ecma-262.htm

Flanagan, D. (2011). *JavaScript: The Definitive Guide* (6 ed.). O'Reilly & Associates.

Google, Inc. (n.d.). *Google Chrome Introduction*. Retrieved from developers.google.com: https://developers.google.com/v8/intro

Kistler, T., & Franz, M. (1999). *A Tree Based Alternative to Java Byte Codes.* M. International Journal of Parallel Programming.

McIlroy, R., & Hodson, O. (n.d.). *Ignition: An Interpreter for V8.* Retrieved from docs.google.com: https://docs.google.com/presentation/d/1OqjVqRhtwlKeKfvMdX6HaCIu9wpZsrzqpIVIwQSuiXQ/edit#slide=id.g1357e6d1a4\_0\_58

McIlroy, R., & Hodson, O. (n.d.). *Ignition: V8 Interpreter.* Retrieved from docs.google.com: https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/edit?ts=56f27d9d#

Mozilla Developer Network. (2014, 4 14). *Rhino overview*. Retrieved from developer.mozilla.org: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/Overview

Mozilla Developer Network. (2014, 12 12). *Tracing JIT*. Retrieved from developer.mozilla.org: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals/Tracing\_JIT

Mozilla Developer Network. (2016, 4 25). *SpiderMonkey Internals*. Retrieved from developer.mozilla.org: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals

Oracle, I. (n.d.). *The Java HotSpot Performance Engine Architecture*. Retrieved from Oracle.com: http://www.oracle.com/technetwork/java/whitepaper-135217.html

Parr, T. (2012). *The Definitive ANTLR 4 Reference.* San Francisco: The Pragmatic Programmers.

PR Newswire. (1995). *Netscape and Sun announce JavaScript.* PR Newswire.

Samsung OSG. (2015, 6). *JerryScript Internals*. Retrieved from http://jerryscript.net/internals/.

Stack Overflow Community. (2010, 1 21). *What are the advantages of just-in-time compilation versus ahead-of-time compilation?* Retrieved from stackoverflow.com: http://stackoverflow.com/questions/2106380/what-are-the-advantages-of-just-in-time-compilation-versus-ahead-of-time-compila

Venners, B. (1996-2017). *Inside the Java Virtual Machine.* Artima, Inc.

Vijayan, K. (2013, 4 5). *The Baseline Compiler Has Landed*. Retrieved from blog.mozilla.org: https://blog.mozilla.org/javascript/2013/04/05/the-baseline-compiler-has-landed/