Advanced Science and Technology Letters
Vol.117 (Information Technology and Computer Science 2015), pp.115-118
http://dx.doi.org/10.14257/astl.2015.117.27

A Study of the JavaScript Compiler and Symbol Table for the Smart Cross Platform

Yunsik Son*, YangSun Lee**

*Dept. of Computer Engineering, Dongguk University
26 3-Ga Phil-Dong, Jung-Gu, Seoul 100-715, KOREA
sonbug@dongguk.edu
**Dept. of Computer Engineering, Seokyeong University
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, KOREA
Corresponding Author : yslee@skuniv.ac.kr

Abstract. Smart Cross Platform (SCP) is virtual machine based solution that supports various programming languages and platforms, and its aims are to support programming languages like ISO/IEC C++, Java and Objective C and smartphone platforms such as Android and iOS. Java Script is a programming language to develop HTML5 contents, and the contents are executed by interpreter which included in web browser unlike compilation based programming and execution methods. In this paper, we will introduce the JavaScript compiler and its symbol table structure for the SCP (Smart Cross Platform) to execute the JavaScript contents by the compilation method.

Keywords: Smart Cross Platform, Virtual Machine, ECMA Script, JavaScript Compiler, Symbol Table, Compiler Construction

1 Introduction

SCP (Smart Cross Platform) is a virtual machine based solution which aims to resolve such problems, and it supports many kind of programming languages like C/C++, Java, and Objective C [1-3]. In this study, a compiler for use in a program designed in the JavaScript programming language to be used on a SCP is designed.

The existing developmental environments for smart devices contents are needed to generate specific target code depending on target devices or platforms, and each platform has its own developing programming language and methodologies. These characteristics in smart contents developments making the contents development process very inefficient. Therefore, even if the same contents are to be used, it must be redeveloped depending on the target machine and a compiler for that specific machine is needed, making.

In order to effectively implement the compiler, we designed the JavaScript compiler by seven modules; lexical analysis, syntax analysis with SDT (Syntax Directed Translation), error recovery, symbol collector, semantic analyzer, code generator, and symbol table.

2 JavaScript Compiler and Symbol Table Model

The JavaScript compiler supports the contents written in JavaScript language on SVM which generates platform independently stack-based SIL code as target code. In this study, the JavaScript to SIL compiler was designed by 7 parts as can be seen in Fig. 1.

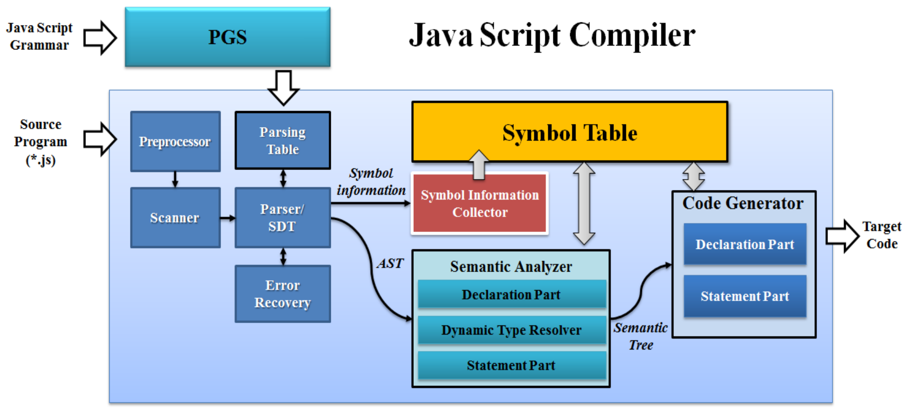


Fig. 1. JavaScript to SIL Compiler Model

The JavaScript to SIL compiler implements the characteristics of the JavaScript language and therefore was designed with seven different parts; Preprocessor, Scanner (lexical analysis), Parser (syntax analysis) with SDT (Syntax Directed Translation), Error recovery, Semantic analyzer, Code generator and Symbol collector and Symbol table. The detailed information for each part is as follows.

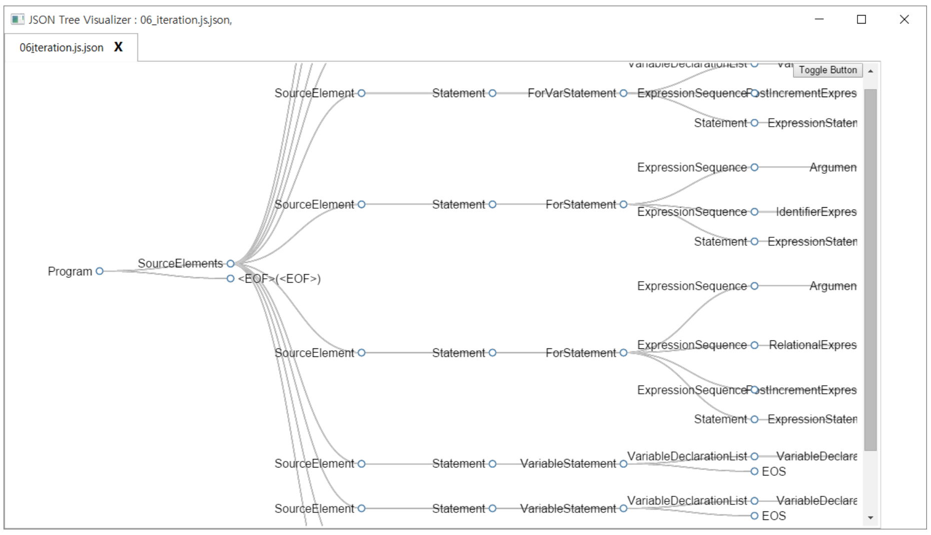
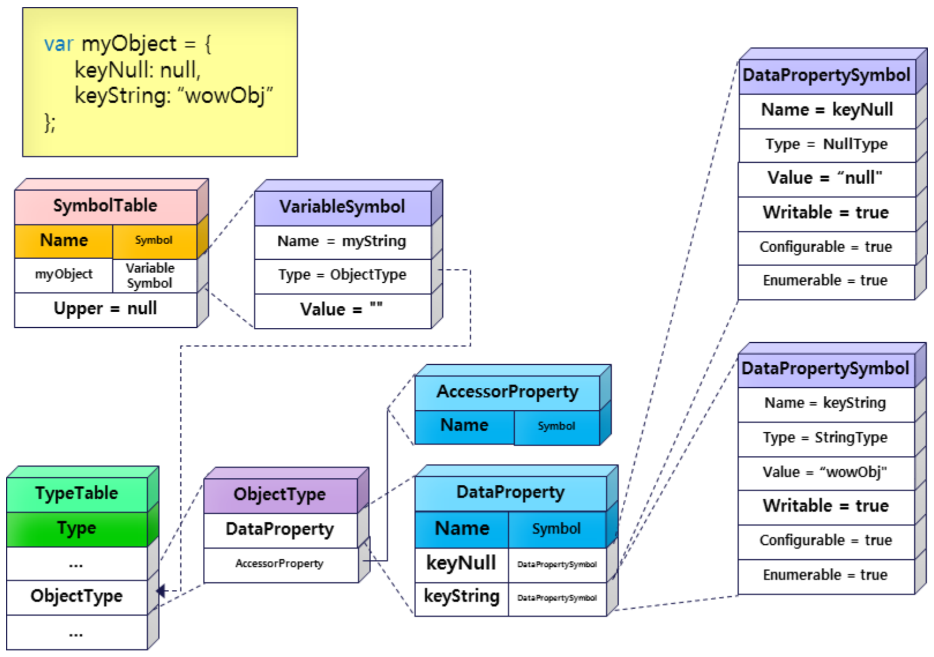


Fig. 2. A visualization of the generated tree structure

Preprocessor, Scanner, Parser with SDT, and Error recovery modules are easily can be group as a processor to analyze the input JavaScript programs and generate analyzed AST (Abstract Syntax Tree) for input program. Tokens and tree nodes are defined based on ECMA Script grammar [4], also a number of defined tokens for lexical structure is 103 and a number of defined tree nodes for syntactic structure is 54. Fig. 2 shows the visualization of the generated tree structure to analyze designed and implemented syntactic structure, AST and front-end of the compiler.

Semantic analyzer is composed by two parts; the declarations semantic analysis module and the statements semantic analysis module. The declarations semantic analysis module checks the process of collecting symbol information on the AST level, to verify cases which are grammatically correct but semantically incorrect. The statements semantic analysis module uses the AST and symbol table to carry out semantic analysis of statements and creates a semantic tree as a result. A semantic tree is a data structure which has semantic information added to it from an AST. It is responsible for all that has not been taken care of during the syntax analysis process and then it is used to generate codes as it has been designed to generate codes easily.

The code generation part receives the semantic tree as an input after all analysis is complete and it generates a SIL code which is semantically equal to the input program (*.js). For this, the SIL code is expressed as symbols so it is convenient to generate and handle them. For type conversion code lists, the same data structure is kept so that the code generation process can take place efficiently. Type conversion code lists are data structures that pre-computed the process of converting a semantic code into a SIL code when generating a code. A code generator visits each nodes of the semantic tree to convert them into SIL codes.



```
var myObject = {
    keyNull: null,
    keyString: "wowObj"
};
```

Fig. 3. A visualization of the generated tree structure

Finally, the symbol collector carries out the job of saving information into the symbol table which is obtained by traversing and analyzing the given AST. The routines consist of the functions, dynamic types, declarations and others. And the symbol table is used to manage the symbols and its attributes in the given programs. The symbol table is designed for easily managing the symbol information and reflected the semantic features of the JavaScript.

The symbol table can be manage types and functions as same object or same structure, and must can be describe the interoperability of all types in JavaScript. Because, the type system of JavaScript is dynamically inferenced at runtime unlike C/C++, Java, and Objective C's static type system. In Fig. 3 shows an example of referencing process for JavaScript's object definition in designed symbol table.

3 Conclusions and Further Researches

In this paper, we have designed a new compiler and symbol table to support the JavaScript programs as the one of the compiler in compiler collection of Smart Cross Platform. We defined seven modules to construct the proposed compiler and to generate a SIL code for use on a SVM which is independent of platforms. In the future, there is need for the semantic analysis and the code generation phases of a JavaScript compiler so that JavaScript base contents can be run on a SVM. Also, further researches are needed that are an optimizer and an assembler for SIL code programs.

Acknowledgments. This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT and Future Planning (No.2013R1A2A2A01067205).

References

1. Y.S. Lee, Y.S. Son, "A Study on the Smart Virtual Machine for Executing Virtual Machine Codes on Smart Platforms", International Journal of Smart Home, SERSC, Vol. 6, No. 4, 2012, Australia, pp. 93-105.
2. Y.S. Lee, Y.S. Son, "A Study on the Smart Virtual Machine for Smart Devices", Information -an International Interdisciplinary Journal, Vol. 16, No. 2, International Information Institute, 2013, Japan, pp.1465-1472.
3. Y. Son, Y.S. Lee, "Design and Implementation of an Objective-C Compiler for the Virtual Machine on Smart Phone", CCIS, Springer, Vol.262, 2011, Heidelberg, pp.52-59.
4. ECMAScript 2015 Language Specification, 6th Edition of ECMA-262, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
5. Y.S. Lee, S.M. Oh, Y.S. Son, "Design and Implementation of HTML5 based SVM for Integrating Runtime of Smart Devices and Web Environments", International Journal of Smart Home, SERSC, Vol.8, No.3, 2014, Australia, pp.223-234.