International Journal of Grid and Distributed Computing
Vol. 11, No. 6 (2018), pp.37-48
http://dx.doi.org/10.14257/ijgdc.2018.11.6.04

A Study on Abstract Syntax Tree for Development of a JavaScript Compiler

Jaehyun Kim and Yangsun Lee*

Dept. of Computer Engineering, Seokyeong University
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, KOREA
**yslee@skuniv.ac.kr*

Abstract

The JavaScript is a programming language for scripting the Web environment, and has become a major language for creating a variety of content, such as Web applications, as the Web environment evolves. The Smart cross platform is a platform that supports multiple programming languages and multiple platforms at the same time. It consists of a set of compilers and a virtual machine based on the intermediate language. We are designing and implementing a virtual machine code based JavaScript compiler for smart cross platform to enable rich JavaScript content to run on the smart cross platform.

The AST is a simple data structure of the tree structure of the input source code. It is easy to process as a program than source code or parse tree, and is used as an intermediate representation in the whole process of the compiler implementation. In this paper, we design and implement the AST for a JavaScript compiler. The input source is transformed into a parse tree by a lexical analyzer and a parser, and the parse tree-to-AST converter implemented by the visitor pattern is circulated and converted into the AST. The AST is designed to be object-oriented and can represent the syntactic structure by only the class structure, and can describe the AST in a consistent way. We also implemented the visitor pattern support for the designed AST to facilitate the implementation of the declaration processor, semantic analyzer, and code generator.

Keywords: *AST(Abstract Syntax Tree), Smart Cross Platform, JavaScript Compiler, Parse Tree-to-AST Converter, Visitor Pattern, Virtual Machine Code, Smart Virtual Machine*

1. Introduction

The JavaScript [1,2] is a programming language for the scripting behavior of the web environment. As the web environment evolves, it becomes a language for creating various contents such as HTML, CSS and various web pages and web applications. Smart Cross Platform [3-5] is a virtual machine system that supports multiple programming languages and multiple platforms at the same time. It consists of a set of compilers and a virtual machine based on the intermediate language. We are designing and implementing a JavaScript compiler for smart cross-platform to enable rich JavaScript content to run on smart cross-platforms.

In this paper, we design and implement the AST for the JavaScirpt compiler [6-8]. The AST is a data structure that removes unnecessary information from the input source code tree structure. It is smaller than an input program or a parse tree and is easy to process, and is used mainly as an intermediate expression throughout the compiler. The JavaScript compiler transforms it into a parse tree through the lexical analyzer and parser generated by the parser generator, and the parse tree-AST converter implemented through the visitor

Received (January 18, 2018), Review Result (March 28, 2018), Accepted (April 4, 2018)
* Corresponding Author

pattern is transformed into the AST. The AST is designed to be object-oriented and can represent the syntactic structure by only the class structure, and can describe the AST in a consistent way. We also implemented the Visitor pattern support for the designed AST to facilitate the implementation of the declaration processor, semantic analyzer, and code generator.

2. Related Studies

2.1. AST(Abstract Syntax Tree)

The AST is a tree that has the same meaning as the input source code and leaves only the necessary information, and is formed by removing unnecessary information from the parse tree in general. The AST contains important information of input source code and is smaller than parse tree, and so it is mainly used as intermediate representation in compiler [9-11].

The parse tree generated by the lexical analyzer and parser in the JavaScript compiler is converted to AST by the parse tree-AST converter. The AST continues to be used as an intermediate representation in place of the input source code in the symbol collector, code generator, and so on.

2.2. Visitor Design Pattern

The Visitor design pattern is a programming pattern that facilitates the addition of functionality to a data class that consists of a tree structure. It allows you to add new functionality that targets the same class without having to modify or recompile the data class by separating the functional class from the data class [12].

In the Visitor pattern, the function class traverses and processes the nodes of the data class and is implemented through the double dispatch of the Accept method of the data class and the Visit method of the Visitor class. Also, depending on the implementation, the Visit method can be configured to collect the processed information from each node of the data class [13].

The JavaScript compiler’s parser supports the Visitor pattern for the parse tree, which implements the parser-AST translator. In addition, AST designed in this paper also supports Visitor pattern, so new processing can be added without modification of intermediate representation.

2.3. ANTLR4 Parser Generator

The ANTLR4 parser generator automatically generates a lexical analyzer and a parser based on the grammar information of the corresponding language using the LL parser generator[14,15]. Unlike ANTLR3, the parser generated by ANTLR4 does not support AST generation but only generates a parse tree. It provides a visitor and listener for the generated parse tree, which makes it easy to process the parse tree. The provided visitor pattern implementation directly traverses child nodes at a desired point in time, and supports return and collection of values in the Visit method.

The ANTLR project provides input grammar information for some known languages [16], and the JavaScript compiler uses ECMAScript syntax to generate lexical analyzers and parsers. We also implemented the parse tree-AST converter using the provided visitor.

2.4. Smart Cross Platform

The Smart Cross Platform [3-5] is a cross-platform environment that supports multiple programming languages on many different types of devices, and is based on the intermediate language, SIL (Smart Intermediate Language). The smart cross platform consists of a compiler set and a smart virtual machine, and applications written in each programming language are translated by the compiler for each language into SIL language, the intermediate language of the smart cross platform. The smart virtual machine for each platform, such as Android and iOS, loads and executes the SIL code program compiled in the intermediate language.

Figure 1 shows the overall structure of the Smart Cross Platform.

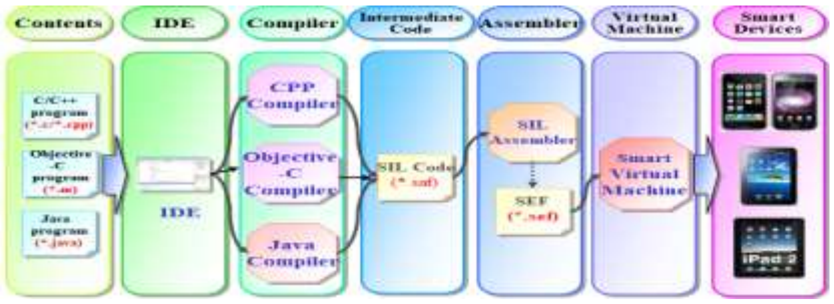


Figure 1. Overall Structure of the Smart Cross Platform

3. AST for a JavaScript Compiler

3.1. JavaScript Compiler

The JavaScript compiler can be divided into parts that generate the AST from the input source code and convert it to SIL, which is the intermediate representation of the smart virtual machine [3-5]. Figure 2 shows the overall structure of the JavaScript compiler.

The part that generates the AST from the input source code consists of a lexical analyzer, a parser, and a parse tree-AST converter. The lexical analyzer separates the input source code into a series of tokens, and the parser generates a parse tree by converting the tokens into a tree structure that conforms to the JavaScript syntax. The parse tree-AST converter removes unnecessary information from the parse tree and converts only the core information to AST. The designed AST supports the Visitor pattern and is defined to be able to work independently of the AST source code in the following process.

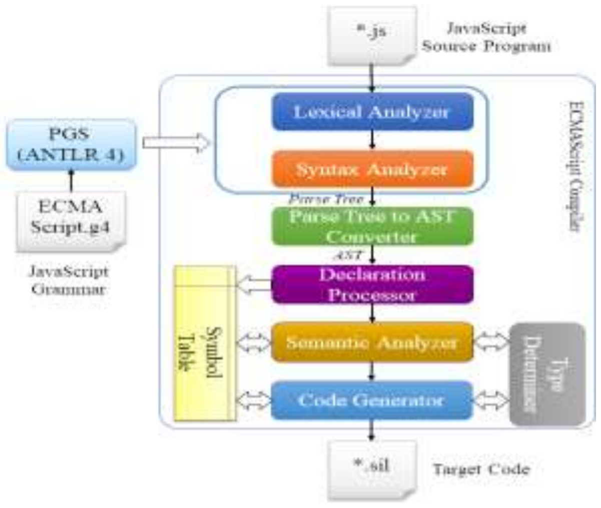


Figure 2. Overall Structure of the JavaScript Compiler

The parts that process the AST and convert to intermediate representation consist of declaration processor, type analyzer, semantic analyzer, and code generator. The declaration processor collects information about declared names of variables, constants, and functions to form a symbol table. The type analyzer analyzes the type of a given expression based on symbol information and AST, and is used in semantic analyzers and code generators. The semantic analyzer parses complex data types and semantics such as function return types, objects generated by constructors, and detects semantically incorrect parts to generate errors. The code generator takes the AST as an input and uses the symbol table and type analyzer to output the target code.

3.2. Generation of a Scanner, Parser

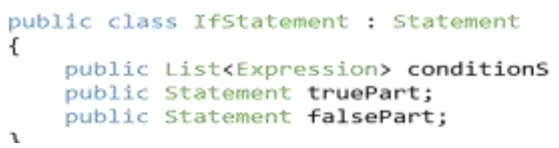
The JavaScript compiler generates and uses a lexical analyzer and a parser using the ANTLR4 parser generator and the JavaScript syntax provided by the project. ANTLR4 supports Visitor and Listener for the generated parse tree, and Visitor supports collection of return values of Visit method. The JavaScript compiler implements the parser-AST translator using Visitor support for the parse tree.

3.3. Design and Implementation of the AST

3.3.1. Object-oriented Design of the AST

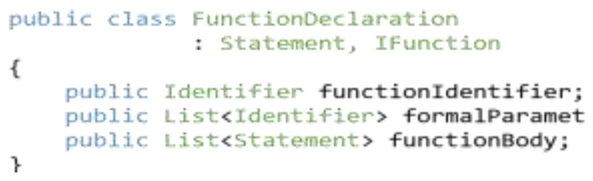
The designed AST (Abstract Syntax Tree) is implemented as a object-oriented method by using irregular heterogeneous AST [9,17], so that the syntax structure can be expressed only by class structure and AST can be described in a consistent way. Irregular heterogeneous AST is one of the AST programming patterns. It is expensive to implement, but it can represent the syntax structure only by its class structure. In the AST, each syntax element is a different data type, and the child elements of each syntax element are represented by the member variables of the data types that can appear in the corresponding syntax element.

Figure 3 is an excerpt of the core part of the IfStatement class, which is the AST class corresponding to the conditional statement. Since conditional statements act as statements, the IfStatement class extends the Statement class. The condition part, true part, and false part of the condition statement consist of the member variables conditionSeq, truePart, and falsePart. The data type of each member variable is declared as the data type of the element that can come in the corresponding syntax element. The condition type is a list of formulas, so the data type of the member variable conditionSeq is List <Expression>, and the true and false parts are statements, so the data types of truePart and falsePart are Statement.



```
public class IfStatement : Statement
{
    public List<Expression> conditionS
    public Statement truePart;
    public Statement falsePart;
}
```

Figure 3. Structure of the IfStatement Class



```
public class FunctionDeclaration
             : Statement, IFunction
{
    public Identifier functionIdentifier;
    public List<Identifier> formalParamet
    public List<Statement> functionBody;
}
```

Figure 4. Structure of the FunctionDeclaration

Figure 4 shows the core part of the AST class in the function declaration. In JavaScript, function declarations are both statements and function objects, so the FunctionDeclaration class extends the Statement class and the IFunction interface. In addition, function identifiers, function argument lists, and function bodies, which are child elements of function declaration statements, are made up of member variables functionIdentifier, formalParameterList, and functionBody, and the data type of each member variable is declared as the data type of the element that can be found in the corresponding syntax element. Since the function identifier can be an identifier, the data type of the functionIdentifier is an identifier, and the identifier of the function argument list can be listed. Therefore, the type of the formalParameterList is List <Identifier>. Also, since the body of the function can contain a list of statements, the data type of functionBody is List <Statement>.

3.3.2. Parse Tree-to-AST Converter using Visitor Patterns

The input source code is converted into a parse tree through a lexical analyzer and a parser, and the parse tree-to-AST converter converts it into an AST. The parser-to-AST translator is implemented as a visitor to the parse tree, and thus consists of the visitor methods corresponding to the part of the parse tree that will be converted to AST.

Figure 5 shows the AST generation process of the parse tree-AST converter.

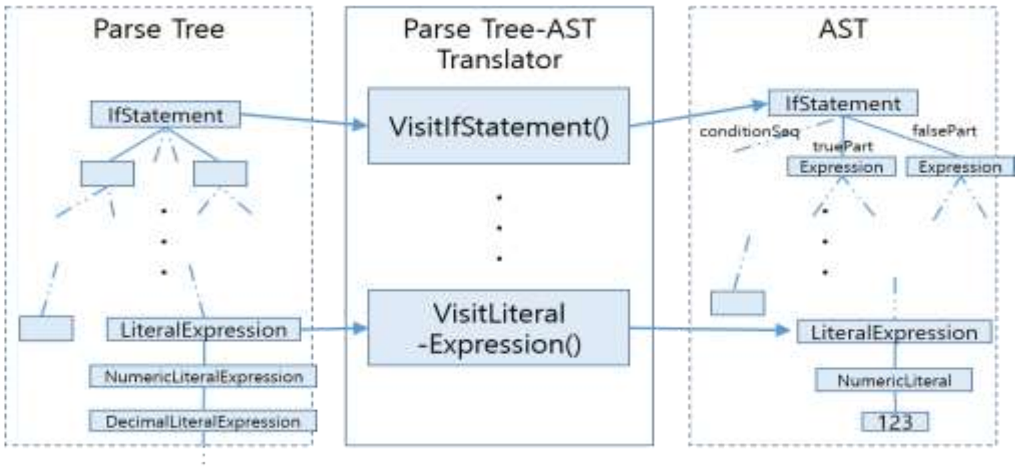
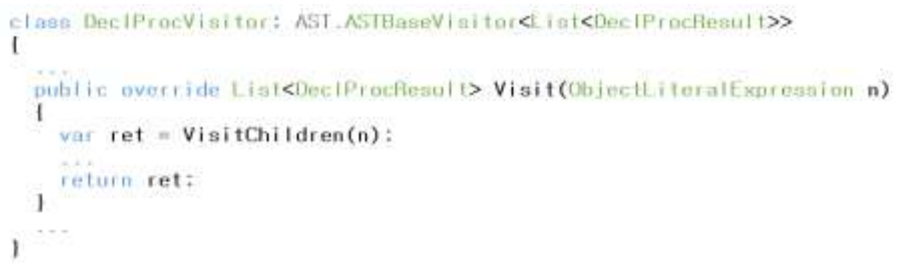


Figure 5. Structure of the Parse Tree-to-AST Converter

3.3.3. Visitor Pattern for the AST

The compiler uses the AST as an intermediate representation instead of the input source code in the subsequent steps after the AST is formed, so the ease of processing of the AST is important. The JavaScript compiler's AST then supports the visitor pattern so that the declaration handler, semantic analyzer, and code generator can effectively traverse and process the AST. The implementation of visitor pattern for this AST can control the traversal point of the child node to a desired point of time, and it is possible to collect the result of each process by setting the returning result of visiting each node.



```
class DeclProcVisitor: AST.ASTBaseVisitor<List<DeclProcResult>>
{
  ...
  public override List<DeclProcResult> Visit(ObjectLiteralExpression n)
  {
    var ret = VisitChildren(n);
    ...
    return ret;
  }
  ...
}
```

Figure 6. Implementation Example of a AST Visitor Pattern

Figure 6 shows an example of using the Visitor pattern implementation supported by AST. To implement a task that traverses and processes the AST, the developer can extend the ASTBaseVisitor class to implement the desired Visitor method. Within each Visitor function, you can call the VisitChild method to control the traversal time of the child nodes. You can set the return value to collect the processing results.

4. Experimental Results and Analysis

4.1. Design of AST

The designed AST consists of two more than fifty concrete classes, three abstract classes, and five interfaces. For each abstract class, there are twenty classes that inherit the Statement, 15 classes that inherit Expression, and seven classes that inherit Literal, and there are nine terminal nodes that do not have AST as a member variable.

Figure 7 is a class that directly inherits the interface and IASTNode used in the designed AST. In the figure, the rectangle with rounded rectangle is the interface, the rectangle is the sphere class, and the rectangle protruding from the right is the terminal node. The IASTNode is the interface that all AST nodes implement, and all AST implement it. In addition to the abstract classes Statement, Expression, and Literal, the classes directly inheriting IASTNode have three non-terminal nodes and two terminal nodes.

Figure 8 shows some of the hierarchical structures of the AST such as the Statement class. Of the rectangles in the figure, Statement, Expression, and Literal are abstract classes. The AST is a structure that starts with the root node Program, and is a child node of the Program. The Expression is a child node of Statement, and Literal is a child node of Expression. FunctionDeclaration inherits Statement, and can also perform as a function and constructor to implement additional IFunction and IConstructor for each role.

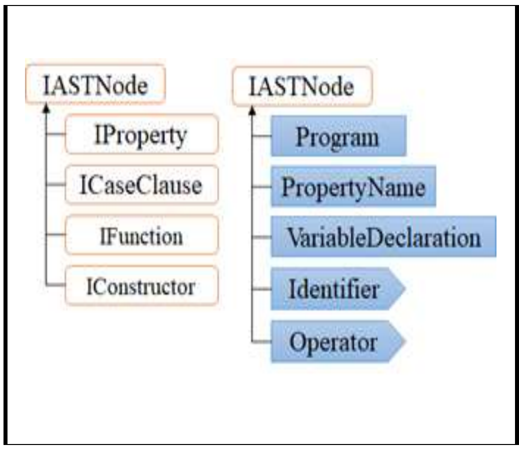


Figure 7. AST Interface and IASTNode

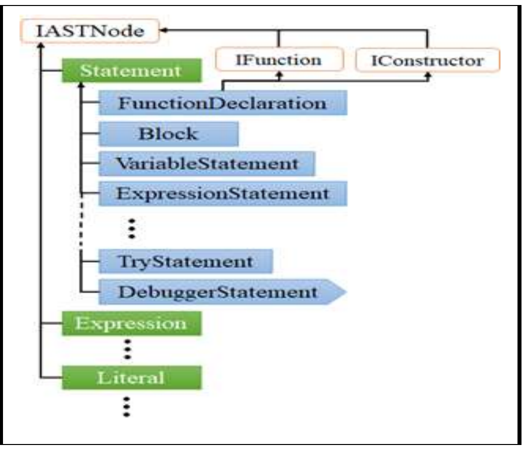


Figure 8. AST Hierarchy of a Statement Class

4.2. Generation of AST

The designed AST can represent the syntax structure only by the class structure, and can be represented in a consistent way, making debugging easy. In Tables 1 and 2, the rectangle is the class and class name corresponding to the AST, and the circle is the value of the terminal node and terminal node. The contents of the connection line represent the member variables and data types of the superclass, the contents written in italics are the data types of the member variables, and the names below are the names of the member variables.

Table 1 shows the structure of the for statement and the generated AST, and the result of converting the generated AST to the JSON visualization result. The root node except the Program node of AST corresponding to the syntax is ForVarStatement, and child node has member variable, and the data type of each child node member variable is data type of AST node that can come to that position. One of the child nodes of the ForVarStatement is a list of formulas, so the corresponding type of the member variable conditionSeq is List<Expression>. In AST, a leaf node does not place a class that extends the IASTNode class as a child member variable. The terminal node corresponding to the number 10 of the code is the NumericLiteral class, and has only the value of the double data type which is the basic data type of the implementation language as a member variable.

Table 1. AST Generation Example for "for statement"

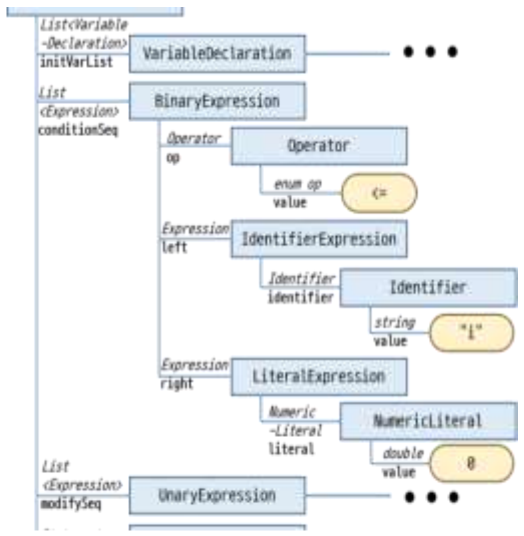
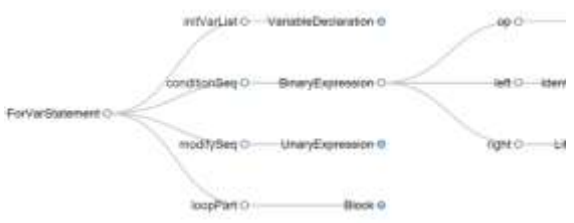
Statement	`for( … ; i <= 10; … ) { …`
AST	 List<Variable-Declaration> initVarList VariableDeclaration List<Expression> conditionSeq BinaryExpression Operator op Operator enum op value <= Expression left IdentifierExpression Identifier identifier Identifier string value "i" Expression right LiteralExpression Numeric-Literal literal NumericLiteral double value 0 List<Expression> modifySeq UnaryExpression
JSON Visualization Result	 initVarList VariableDeclaration conditionSeq BinaryExpression ForVarStatement modifySeq UnaryExpression loopPart Block op left ident right Lit

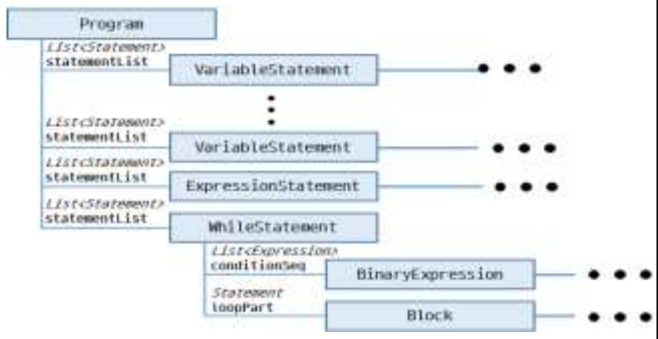
Table 2 is a visualization of the structure of some of the ASTs generated by the JavaScript program that calculates decimals, and the result of converting the generated AST to the JSON visualization result. The root node of the AST corresponding to the entire program is a Program node, and there are six Statements, one ExpressionStatement, and one WhileStatement as AST elements corresponding to child nodes in the Program node's member variable statementList. The class corresponding to the program's core algorithm is WhileStatement, which consists of the member variables conditionSeq and loopPart. The data type of the member variable conditionSeq is List<Expression>, the BinaryExpression object is assigned as a child node, the data type of the member variable loopPart is Statement, and Block is assigned as a child node.

Table 2. AST Generation Example for " Prime Number"

Program

```
var max = 100;

var i = 0;
var j = 0;
var k = 0;
var rem = 0;
var prime = 0;

i = 2;

while (i <= max) {
    prime = 1;

    k = i / 2;
    j = 2;
    while (j <= k) {
        rem = i % j;
        if (rem == 0) prime = 0;
        j = j + 1;
    }
    if (prime == 1) printf("%d\n", i);
    i = i + 1;
}
```

AST



JSON Visualization Result

4.3. AST Visitor

Using the Visitor pattern implemented in the JavaScript compiler's AST, you can easily traverse the AST, determine the traversal time of the child nodes directly, and collect and collect the results processed by each Visitor method. Table 3 shows the function of converting AST to JSON string by using a Visitor pattern.

Table 3. AST Generation Example for “ Prime Number”

```
class AstToJsonVisitor:
ASTBaseVisitor<StringBuilder>
{
   protected override StringBuilder
   AggregateResult(StringBuilder current,
      StringBuilder next) {
         ...
         return current.Append(", ").Append(next);
      }
      ...
```

```
      public override StringBuilder Visit(ForVarStatement n)
      {
         StringBuilder sb = new StringBuilder();
         sb.AppendLine("{\"name\": \"ForVarStatement\","
              + " \"kind\": \"node\"," +" \"member\": [");
         sb.AppendLine("{\"name\": \"initVarList\","+" \"kind\": \"member\","
              + " \"type\": \"List<VariableDeclaration>\","+ " \"node\": [");
         if  (n.initVarList != null) {
              sb.Append(Visit(n.initVarList));
         }
         sb.AppendLine("]}");
         sb.AppendLine(",");
         ...
         If  (n.loopPart != null) {
              sb.Append(Visit(n.loopPart));
         }
         sb.AppendLine("]}");
         sb.AppendLine("]}");
         return sb;
      }
      ...
}
```

JSON with AST converted

```
{"name": "ForVarStatement", "kind": "node", "member": [
     {"name": "initVarList", "kind": "member",
       "type": "List<VariableDeclaration>", "node": [... ]},
     {"name": "conditionSeq", "kind": "member","type": "List<Expression>",
       "node": [{"name": "BinaryExpression", "kind": "node","member": [... ]}
     ]},
     {"name": "modifySeq", "kind": "member", "type":"List<Expression>",
       "node": [... ]},
     {"name": "loopPart", "kind": "member", "type":"Statement", "node": [
       ...
     ]}
]}
```

The Visitor class AstToJsonVisitor for this function extends ASTBaseVisitor. In addition, the generic class of the return value was extended to StringBuilder to return and collect the result processed by the Visitor method corresponding to each node, and the AggregateResult function was overridden to merge the two strings in the process of collecting the return value. the Visitor method is overrided to convert to a string for each node. Then, in each overridden method, the result of JSON transformation of the child node is called VisitChildren and delegated to the Visitor method of the child node and merged with its own JSON transformation result.

5. Conclusions and Further Researches

In this paper, we designed and implemented the AST for a JavaScript compiler. The AST designed in this paper is an object-oriented design that can represent the syntax structure only by class syntax structure and can describe the AST in a consistent way. In addition, the AST is generated by converting the parse tree into the AST through the parse tree-to-AST converter implemented through the visitor pattern, and the designed AST also supports the visitor pattern to facilitate the circulation and processing of the AST. Using the AST and Visitor pattern implemented in this paper, we will implement the next step of the JavaScript compiler, the semantic analyzer and the code generator.

Acknowledgements

This research was supported by Seokyeong University in 2017.

References

[1] D. Crockford, “JavaScript: The Good Parts”, O'Reilly Media, **(2008)**.
[2] J. Thompsons, “JavaScript”, Createspace Independent Pub, **(2017)**.
[3] Y. Lee and Y. 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)**, pp. 93-105.
[4] Y. Lee and Y. Son, “A Study on the Smart Virtual Machine for Smart Devices”, Information -an International Interdisciplinary Journal, International Information Institute, Japan, vol. 16, no. 2, **(2013)**, pp. 1465-1472.
[5] S. Han, Y. Son and Y. Lee, “Design and Implementation of the Smart Virtual Machine for Smart Cross Platform”, Journal of Korea Multimedia Society, vol. 16, no. 2, **(2013)**, pp. 190-197.
[6] Y. Son, J. Kim and Y. Lee, “A Study on the JavaScript Compiler for Extension of the Smart Cross Platform”, Advanced Science and Technology Letters, vol. 106, **(2016)**, pp. 52-55.
[7] Y. Son, S. Oh and Y. Lee, “A Symbol Table Verification Method for JavaScript Compiler using Reverse Translator on HTML5 Smart Virtual Machine”, Information, International Information Institute, vol. 20, no. 5(A), **(2017)**, pp. 5401-5408.
[8] ECMAScript 2015 Language Specification, 6th Edition of ECMA-262, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.
[9] A. B. Tucker and R. E. Noonan, “Programming Languages”, 2nd Edition, McGraw-Hill Education, **(2006)**.
[10] D. Grune, H. E. Bal, C. J. H. Jacobs and K. G. Langendoen, “Modern Compiler Design”, John Wiley & Sons, **(2000)**.
[11] Y. Son and Y. Lee, “A Study on the Semantic Analyzer of an Objective-C Compiler based on AST-Semantic Tree Transformation”, Information, International Information Institute, vol. 17, no. 3, **(2014)**, pp. 1077-1082.
[12] E. Gamma, R. Helm, R. Jhonson and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, USA, **(2002)**.
[13] P. Buchlovsky and H. Thielecke, “A Type-theoretic Reconstruction of the Visitor Pattern”, Electronic Notes in Theoretical Computer Science, vol. 155, **(2006)**, pp. 309-329.
[14] T. Parr, ANTLR 4, http://www.antlr.org/.
[15] T. Parr, The Definitive ANTLR 4 Reference, Pragmatic Bookshelf, **(2007)**.
[16] Antlr Project, grammars-v4, https://github.com/antlr/grammars-v4.
[17] T. Parr, “Language Implementation Patterns”, Pragmatic Bookshelf, **(2009)**.

Authors

JaeHyun Kim, he received the B.S. degree from the Dept. of Mathematics, Hanyang University, Seoul, Korea, in 1986, and M.S. and Ph.D. degrees from Dept. of Statistics, Dongguk University, Seoul, Korea in 1989 and 1996, respectively. He was a chairman of Dept. of Internet Information 2002-2007. Currently, he is a member of the Korean Data & Information Science Society and a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include mobile programming, cloud system and big data analysis.

Yangsun Lee, he received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 1985, and M.S. and Ph.D. degrees from Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 1987 and 2003, respectively. He was a Manager of the Computer Center, Seokyeong University from 1996-2000, a Director of Korea Multimedia Society from 2004-2018, a General Director of Korea Multimedia Society from 2005-2006, a Vice President of Korea Multimedia Society in 2009, and a Senior Vice President of Korea Multimedia Society in 2015-2016. Also, he was a Director of Korea Information Processing Society from 2006-2014 and a President of a Society for the Study of Game at Korea Information Processing Society from 2006-2010. And, he was a Director of HSST from 2014-2018. Currently, he is a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.