International Journal of Database Theory and Application
Vol.10, No.7 (2017), pp.27-40
http://dx.doi.org/10.14257/ijdta.2017.10.7.03

Design and Implementation of the Symbol Table for Object-Oriented Programming Language

Yangsun Lee

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

Abstract

The symbol table used in the existing compiler stores one symbol information into a plurality of sub tables, and the abstract syntax tree necessary for generating symbols has a binary tree structure composed of a single data structure node. This structure increases the source code complexity of modules that generate symbols and modules that reference symbol tables, and when designing a compiler for a new language, it is necessary to newly design an abstract syntax tree and a symbol table structure considering the characteristics of the language.

In this paper, we apply the object-oriented principle and visitor pattern to improve the abstract syntax tree structure and design and implement the symbol table for the object - oriented language. The design of AST (abstract syntax trees) with object-oriented principles and Visitor patterns reduces the time and cost of redesign because it makes it easy to add features of the language without the need to redesign the AST (abstract syntax tree) for the new object-oriented language. In addition, it is easy to create a symbol through the Visitor pattern. Symbol tables using the open-close principle and the dependency inversion principle can improve the code reusability of the source code that creates and refer to the table and improve the readability of the code.

Keywords*: Symbol Table, Compiler, Object-oriented Principle, Visitor Pattern, AST*

1. Introduction

A compiler is a system software that translates user-written programs into machine-readable instructions that can be executed by a computer. Because of the nature of the compiler, the development of a new compiler is required whenever the language and the platform are different, various studies have been conducted on compiler modularization. Despite these researches, however, the symbol table for storing and managing symbol information was made up of a number of sub-tables and was a complicated structure referring to each other. In addition, since the abstract syntax tree used to represent the characteristics of the language and generate symbols is a binary tree structure representing all kinds of nodes with a single data structure, it is necessary to change the overall structure of the tree when a new language feature is added, There has been a problem that a whole modification of the routines using the tree is necessary. Symbol generation process, which collects and completes symbol information, also analyzes the abstract syntax tree through complex conditional statements and recursive calls, so the complexity of the compiler source code is still high and maintainability is poor [1,2].

In order to solve this problem in this paper, we have improved the structure of the abstract syntax tree by applying the object-oriented principle and Visitor pattern to easily maintain the characteristics of the existing object-oriented language and to easily add new characteristics. In order to reduce the complexity of reference routines, we designed and implemented a symbol table based on object-oriented principles. Improved abstract syntax trees make it easy to add properties without modifying existing tree structures when

ISSN: 2005-4270 IJDTA

creating a tree for a language of the same object-oriented language. The process of generating the symbol using the object-oriented principle traverses the abstract syntax tree node through the Visitor class, performs semantic analysis to collect symbol information without additional conditional statements, and inserts the symbol information into the symbol table. Since it is expressed by only two tables, it is possible to reduce the source code complexity of the process of generating a symbol and the process of referring to a symbol table.

2. Researches

2.1. AST (Abstract Syntax Tree)

The compiler expresses the program in a tree structure in order to express the meaning of the input program more effectively, and it is easily reconfigured according to the characteristic of the code. Parse tree is one of these tree representation methods. However, because the parse tree contains a lot of unnecessary redundant information, it is inappropriate for intermediate languages and requires a more efficient method. Therefore, the abstract syntax tree (AST) is an expression method that removes unnecessary information from the parse tree. The AST node is divided into a terminal node and a non-terminal node. A terminal node represents a constant or a variable, and a non-terminal node represents an AST operator. The operands of one operator are bound to child nodes and are organized in a subtree form with their operator nodes. Figure 1 shows the Figure 1 shows an example of an AST for the integer type variable x.

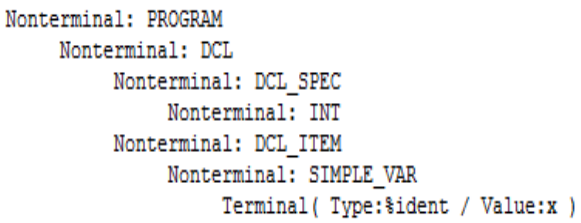

Figure 1. Example of an AST for the Integer Type Variable x

2.2. Visitor Pattern

Visitor patterns, one of the software design patterns, allow you to add new behavior to the object structure without modifying the existing structure by separating the structure and function of the object. It is designed in such a way that a separate visitor class is created by collecting the functions of each class and each object is circulated by the visitor class. The visitor pattern is suitable for cases where the structure of the object does not change well, and is also suitable when additional functions are required in the future. Regardless of the type or structure of the object, you can use the Visitor object to perform the function anywhere you have an interface for the visit. Figure 2 shows the Visitor class of the visitor pattern visiting each element.

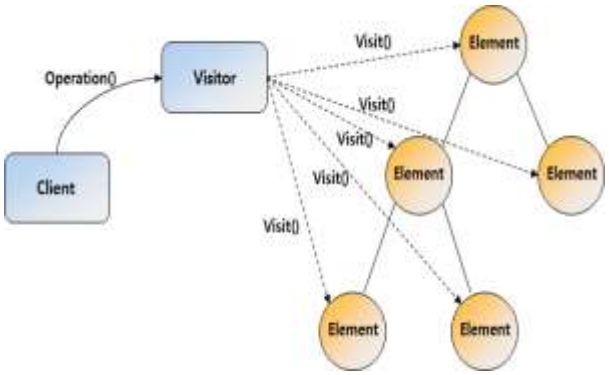

Figure 2. Visitor Pattern Visiting Each Element

2.3. Code Metrics

A code metrics is a measure of the quality of source code quality. It includes maintenance index, class linkage, inheritance level, number of lines of code, and halstead metrics. A maintenance index is a measure of the relative ease of maintenance of a code with a value between 0 and 100. Numerical values are calculated using the number of lines of code, program volume, and cyclic complexity. The larger the value, the more convenient the maintenance work. The color value can be set according to the numerical value, and the ease of maintenance of the code can be visually judged.

Class coupling metrics measure the degree of association of classes by parameters in a class, local variables, return types, method invocations, and interface implementations. The lower the degree of class coupling, the higher the cohesion, which means that the higher the degree of coupling, the more difficult it is to maintain the source code. Inheritance level refers to the number of class definitions that extend to the root in the class hierarchy. The more complex the hierarchy, the more difficult it is to determine where certain methods and fields are defined or redefined.

The number of lines of code is a measure of the number of lines in the source code, which tells you how much work is done on the method, and the higher the number, the more difficult it is to maintain. Circular complexity is a metric developed by McCabe in 1976 that measures the complexity by the number of paths based on a control flow graph that represents the logic flow of the program. It is easy to compare different programs by measuring the complexity of the source code and presenting the result as a single value. Numbers are calculated based on the number of control statements.

Halstead metrics is an index of quantitative complexity based on the number of operators and operands in source code. It was introduced by Maurice Howard Halstead in 1977. Halstead metrics is a program that measures the program length, the number of program vocabulary, the program volume, the program's level of understanding of the program and its maintainability, and the effort required to understand and implement the program code. , And method and type, and the number of estimated bugs, which is the estimated value of the number of bugs that can occur in the package.

To compare the analysis and improvement of the symbol table structure to which the object-oriented principle is applied, a code metric based on the control flow of the code is required. Typical metrics include a halstead metrics and cyclic complexity. Halsted metrics can obtain maintenance-specific values that can’t be calculated by cyclic complexity, but they are difficult to use in the program design stage because they are calculated based on the number of operators and operands. Also, unlike the cyclic complexity, which shows the complexity of the control flow, the control flow is not reflected, so the halstead metrics can be high even in source code without conditional statements. Therefore, we use the cyclomatic complexity based on control flow for analysis and evaluation of symbol table source code.

3. AST Structure Improvement

The structure of the AST (Abstract Syntax Tree) node used in the existing compiler expresses information of all terminals and non-terminal nodes as one structure. In this structure, the analysis module using the AST can classify the node type through the attribute of the node, and the analysis process according to the type can be performed. Therefore, the maintenance complexity of the source code is increased and the maintainability is degraded. The complexity of the AST analysis process affects the symbol generation process because of the structure of the compiler that generates symbols through the analysis of AST. To solve this problem, we designed the node structure of AST as layered object structure according to object orientation principle.

The information of the AST node is represented as a class object. The hierarchical structure of the node is the highest class, and there exists the AST Node class and the non-terminal Node class and the terminal node class which inherit it. Figure 3 shows the hierarchy of the AST nodes.

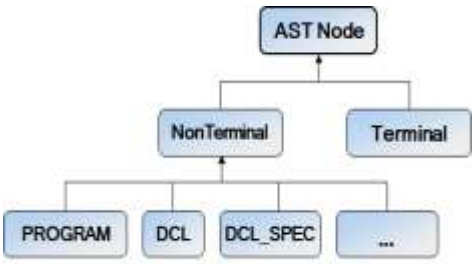

Figure 3. Hierarchy Structure of the AST Nodes

The terminal node class represents the terminal node information, and the node corresponding to the variable and constant information is defined by the corresponding class. The non-terminal node class classifies non-terminal nodes into child classes that represent information of non-terminal nodes. In this way, the node class classified according to each node information has a structure suitable to the principle of single responsibility. In addition, the AST Node class, which is the best parent class of a node, is defined as an abstract class so that it can observe the polymorphism and the liskov substitution principle in the semantic analysis step of analyzing the AST structure. The improvement of the AST structure does not require any additional cost since only the type of node generated in the same manner as the existing generation routine is changed into the hierarchical class without changing the generation routine.

Figure 4 shows an example of the layered node structure of the AST structure in Figure 2. The non-terminal node indicates the type of the non-terminal node by the name of the node, and the terminal node class indicates 'T:' meaning terminal, and the corresponding class is the terminal node class.

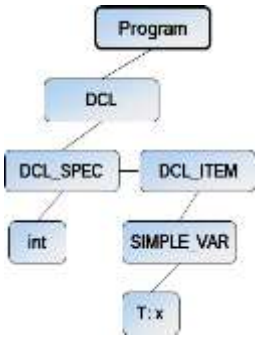

Figure 4. Layered Node Structure of the AST Structure

4. Symbol Table Design

We design a symbol table structure using object-oriented principles and design a hierarchical symbol class to store symbols according to the symbol table to which this principle applies. In addition, a type class for storing the necessary type information in the symbol is designed and a type table for managing the type object is designed.

4.1. Hierarchical Design of Symbols

The symbol table stores hierarchical symbol objects to obtain symbol information with minimum conditions. The symbol class is designed as a hierarchical structure that has the abstract class Symbol as its top class and has Variable, Field, Function, Method, and Parameter as subclasses. Unlike the existing symbol table structure, which stores symbols in multiple tables, the symbol table to which the object-oriented principle is applied stores symbol attributes as one of the subclasses of the symbol class in the hierarchical structure.

Figure 5 shows the hierarchy of symbols. Each symbol subclass derived from Symbol class is a class for a node that represents the characteristics of a symbol among non-terminal nodes. The nodes corresponding to the variable information are subclasses inheriting the Variable class, and the nodes having the function and method information inherit the Function and Method classes.

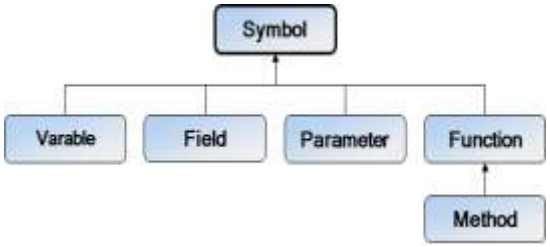

Figure 5. Hierarchical Symbol Table Structure

4.2. Symbol Table Structure

The symbol table stores the symbol object as a value with the symbol name as a key. The attributes of a symbol are stored in a symbol defined as a subclass type of a symbol hierarchy applying the object-oriented principle, and symbol type information is referenced in a type table. When referring to a symbol table from the outside, a symbol is searched based on the symbol name. Since it is made up of objects corresponding to each kind of symbols, additional analysis of symbols is not necessary. Figure 6 shows the structure in which the symbol table has symbol names and symbol values.

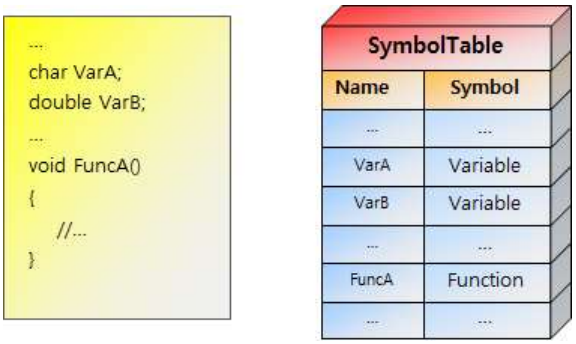

Figure 6. Symbol Table Structure

4.3. Hierarchical Design of Type

The type table is a table for managing symbol type information. The type is designed as a hierarchical structure having the abstract class Type as the top class as the symbol. Primitive, which is a subclass of Type, is a general type class such as Int and Char. It is composed of Reference class, which is a reference type, and UserDefined class, which is a user defined type such as Typedef, Interface, and Struct. Figure 7 shows the type hierarchy.

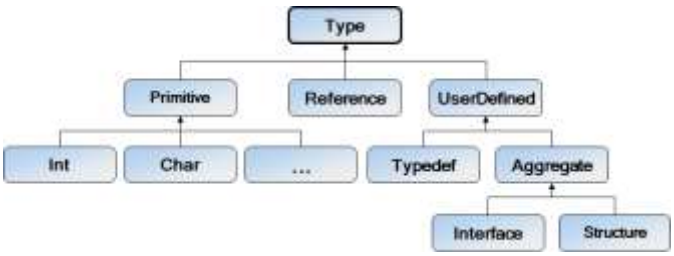

Figure 7. Type Hierarchy Structure

4.4. Type Table Structure

The type table stores the name of the type as a key and the type object as a value. The type attribute is stored in a hierarchical type, and the type information stored in the type table is referred to when referring to the symbol type. If multiple types exist in one type information, such as a typedef, the type information is linked and managed in a list form. Figure 8 shows the structure for storing basic type objects in a type table.

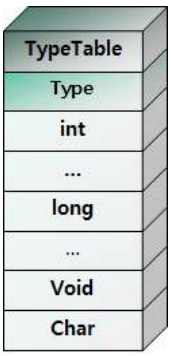

Figure 8. Type Table Structure

4.5. Symbol Generator

The symbol generator traverses the abstract syntax tree through the abstract syntax tree visitor class applying the visitor pattern structure, collects symbol information, inserts the symbol into the symbol table, and completes the symbol. Figure 9 shows the structure of the symbol generator.

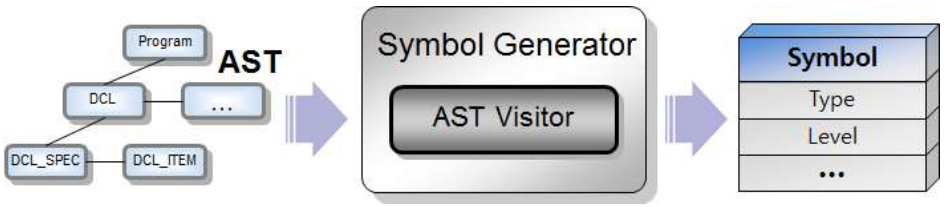

Figure 9. Symbol Generator Structure

The structure of the abstract syntax tree visitor is divided into the visitor generation class that traverses the tree and the node visit, and the symbol generation process that collects the semantic analysis functions of the abstract syntax tree nodes. The function of abstract syntax tree visitor is divided into tree traversal and node visit. First, the tree traversal traverses the abstract syntax tree received as input. Next, the node visit visits the node that is encountered during the tree traversal, and performs the symbol generation process of the visited node through the visitor to collect symbol information from the node. Figure 10 shows how a visitor visits an abstract syntax tree node and processes the node.

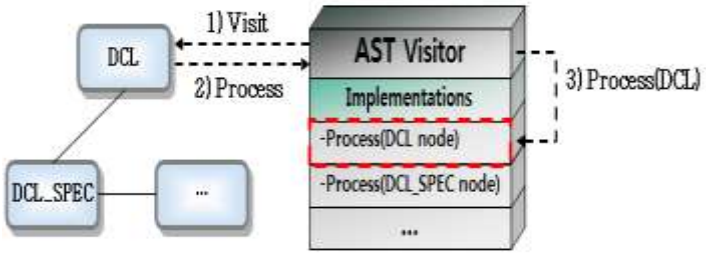

Figure 10. AST Node Processing through Visitor

The node visiting the abstract syntax tree traverses the symbol generation process through the visitor and collects the symbol information of each node. It collects information such as name, type, level, initial value and so on necessary to construct the symbol, and completes one symbol. The completed symbol is inserted into the symbol table.

4.6. Structure of Symbol Table

The symbol table structure using the proposed symbol table is as follows. Figure 11 shows the management structure of a single type symbol. In the symbol table, VarA, which is a variable name, acts as a key of a symbol table. A Variable object stored as a value refers to Char type, which is a type of VarA, in the type table, Const information indicating a constant type variable, whether or not to initialize, and the like are stored in the object.The type table handles the basic type as an object in the same way as the symbol table, and it shows the storing of Char and int type information used in VarA and VarB.

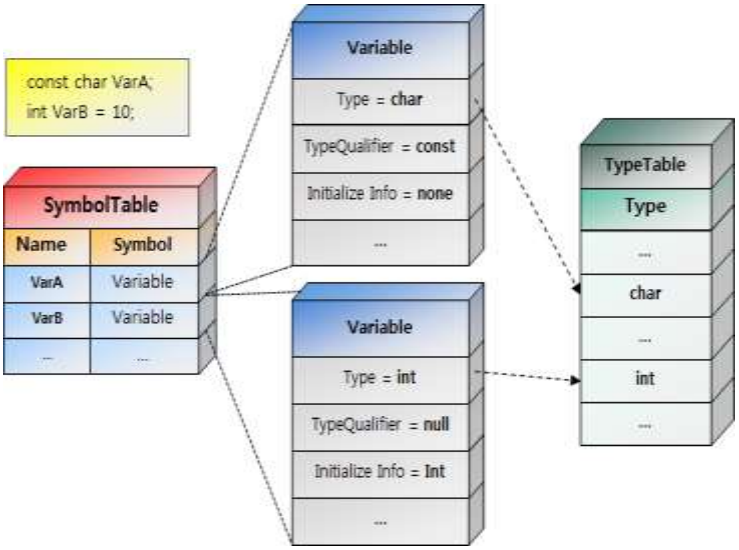

Figure 11. Management Structure of a Single Type Symbol

4. Experimental Results and Analysis

First, we confirmed that a compiler applying the proposed AST structure for CPP, Objective C, and Java language input programs having the same meaning normally generates AST. Example 1 shows the AST generation for the CPP source code, example 2 shows the AST generation for the Objective C source code, and finally example 3 shows the AST generation for the Java source code.

CPP Source Code	CPP AST

class Point
{
  double x, y;
public :
  void set(double x, double y)
  {
  this->x = x; this->y = y;
  }
  void showPoint()
  {
  std::cout << "(" << x << ", "<< y << ")"<<endl;
  }
};
 	
class ColorPoint : public Point
{
  char* color;
public:
  void setColor(char* color)
  {
  this->color = color;
  }
  void showColorPoint()
  {
  cout << color << ":";
  showPoint();
  }
};
 	
void main()
{
  Point p;
  ColorPoint cp;
  cp.set(3,4);
  cp.setColor("Red");
  cp.showColorPoint();
}
	Nonterminal: PROGRAM
	  Nonterminal: DCL
	  Nonterminal: DCL_SPEC
	  Nonterminal: CLASS_DCL
	  Nonterminal: CLASS_HEAD
	  Terminal(Type:id / Value:Point)
	  Nonterminal: MEMBER_DCL
	  Nonterminal: DCL_SPEC
	  Nonterminal: DOUBLE
	  Nonterminal: DCL_ITEM
	  Nonterminal: SIMPLE_VAR
	  Terminal(Type:id / Value:x)
	  Nonterminal: DCL_ITEM
	  Nonterminal: SIMPLE_VAR
	  Terminal(Type:id / Value:y)
	  Nonterminal: PUBLIC
	  Terminal(Type:tpublic / Value:public)
	  Nonterminal: MEMBER_FUNC_DEF
	  ...
	  Nonterminal: FUNC_DEF
	  Nonterminal: DCL_SPEC
	  Nonterminal: INT
	  Nonterminal: FUNC_DCL
	  Nonterminal: SIMPLE_VAR
	  Terminal(Type:id / Value:main)
	  Nonterminal: COMPOUND_ST
	  Nonterminal: DCL
	  Nonterminal: DCL_SPEC
	  Nonterminal: TYPE_NAME
	  Nonterminal: CLASS_TAG
	  Terminal(Type:className / Value:Point)
	  Nonterminal: DCL_ITEM
	  Nonterminal: SIMPLE_VAR
	  Terminal(Type:id / Value:p)
	  ...

Example 1. AST Generation for the CPP Source Code

Objective C Source Code	Objective C AST

//Point.h
@interface Point : NSObject
@property (readwrite) double _x;
@property (readwrite) double _y;
-(void) showPoint;
@end;
 	
//point.m
@implemantation Point
@synthesize _x;
@synthesize _y;
-(void) showPoint
{
	Nonterminal: PROGRAM
	  Nonterminal: CLASS_INTERFACE
	  Terminal(Type:%ident / Value:Point)
	  Nonterminal: INTERFACE_DCL_LIST
	  Nonterminal: PROPERTY_DCL
	  Terminal(Type:@property / Value:@property)
	  Nonterminal: PROPERTY_ATTRIBUTES
	  Nonterminal: SIMPLE_PROPERTY_ATTRIBUTE
	  Terminal(Type:%ident / Value:readwrite)
	  Nonterminal: DCL
	  Nonterminal: DCL_SPEC
	  Nonterminal: DOUBLE
	  Nonterminal: DCL_ITEM
	  Nonterminal: SIMPLE_VAR
	  Terminal(Type:%ident / Value:_x)
	  Nonterminal: PROPERTY_DCL
	  Terminal(Type:@property / Value:@property)


```
            NSLog(@"(%lf, %lf)\n",_x, _y);
}
@end

//ColorPoint.h
@interface ColorPoint : Point
@property (copy) NSString* _color;
-(void) showColorPoint;
@end

//ColorPoint.m
@implemantation ColorPoint
@synthesize _color;
-(void) showColorPoint
{
            NSLog(@"%@ : ",_color);
            [super showPoint];
}
@end

//main.m
void main (int argc, const char * argv[])
{
            Point *p = [[Point alloc] init];

            ColorPoint *cp = [[ColorPoint alloc] init];
            cp._x = 3.1;
            cp._y = 4.5;
            cp._color = @"Black";

            [cp showColorPoint];
}
```

```
                        ...
Nonterminal: FUN_DEF
    Nonterminal: DCL_SPEC
        Nonterminal: INT
    Nonterminal: FUNC_DCL
        Nonterminal: SIMPLE_VAR
            Terminal( Type:%ident / Value:main )
        Nonterminal: PARAM_DCL
            Nonterminal: INT
            Nonterminal: SIMPLE_VAR
                Terminal( Type:%ident / Value:argc )
        Nonterminal: PARAM_DCL
            Nonterminal: CONST
            Nonterminal: CHAR
            Nonterminal: POINTER
            Nonterminal: ARRAY_VAR
                Nonterminal: SIMPLE_VAR
                    Terminal( Type:%ident / Value:argv )
    Nonterminal: COMPUND_ST
        Nonterminal: DCL
            Nonterminal: DCL_SPEC
                Nonterminal: CLASS
                    Terminal( Type:%class_name / Value:Point )
            Nonterminal: DCL_ITEM
                Nonterminal: POINTER
                Nonterminal: SIMPLE_VAR
                    Terminal( Type:%ident / Value:p )
                Nonterminal: MESSAGE_EXP
                    Nonterminal: RECEIVER_PART
                        Nonterminal: MESSAGE_EXP
                            Nonterminal: RECEIVER_PART
                                ...
```

Example 2. AST Generation for the Objective C Source Code

Java Source Code	Java AST

```
class Point
{
   private double x;
   private double y;
   Point()
   {
       this.x = 0.0;
       this.y = 0.0;
   }
   Point(double x, double y)
   {
       this.x = x;
       this.y = y;
   }
   public double getX()
   {
       return x;
   }
   public double getY()
   {
       return y;
   }
   public String toString()
   {
       return "(" + x + ", " + y + ")";
   }
}
class ColorPoint extends Point
{
   private StringBuffer color = new StringBuffer();

   ColorPoint()
   {
       super();
```

```
Nonterminal: PROGRAM
    Nonterminal: CLASS_DCL
        Terminal: Point
        Nonterminal: CLASS_BODY
            Nonterminal: FIELD_DCL
                Nonterminal: PRIVATE
                Nonterminal: DCL_SPEC
                    Nonterminal: DOUBLE_TYPE
                Nonterminal: VAR_ITEM
                    Nonterminal: SIMPLE_VAR
                        Terminal: x
            Nonterminal: FIELD_DCL
                Nonterminal: PRIVATE
                Nonterminal: DCL_SPEC
                    Nonterminal: DOUBLE_TYPE
                Nonterminal: VAR_ITEM
                    Nonterminal: SIMPLE_VAR
                        Terminal: y
                            ...
    Nonterminal: CLASS_DCL
        Nonterminal: PUBLIC
        Terminal: javaClassTest
        Nonterminal: CLASS_BODY
            Nonterminal: METHOD_DCL
                Nonterminal: METHOD_HEAD
                    Nonterminal: PUBLIC
                    Nonterminal: STATIC
                    Nonterminal: METHOD_ITEM
                        Terminal: main
                        Nonterminal: PARAM_DCL_LIST
                            Nonterminal: PARAM_DCL
                                Nonterminal: DCL_SPEC
                                    Nonterminal: ARRAY_TYPE
                                        Nonterminal: SIMPLE_NAME
                                            Terminal: String
                                Nonterminal: SIMPLE_VAR
                                    Terminal: args
                Nonterminal: METHOD_BODY
                    Nonterminal: BLOCK (start : 46  end : 52)
```

```
        color.append("White");
    }
    ColorPoint(double x, double y, String newColor)
    {
        super(x, y);
        color.append(newColor);
    }
    public StringBuffer getColor()
    {
        return color;
    }
    public String toString()
    {
        return getColor() + super.toString();
    }
}
public class javaClassTest
{
    public static void main(String[] args)
    {
        ColorPoint a = new ColorPoint(2.1, 4.1, "Black");
        ColorPoint b = new ColorPoint();

        System.out.println("ColorPoint a: " + a);
        System.out.println("ColorPoint b: " + b);
    }
}
```

```
Nonterminal: LOCAL_VAR_DCL
  Nonterminal: DCL_SPEC
    Nonterminal: CLASS_INTERFACE_TYPE
      Nonterminal: SIMPLE_NAME
        Terminal: ColorPoint
  Nonterminal: VAR_ITEM
    Nonterminal: SIMPLE_VAR
      Terminal: a
    ...
```

Example 3. AST Generation for the Java Source Code

Since the basic object information of each language has the same meaning, the AST of each object-oriented language can be expressed by extending the specific nodes having different differences in the language in the proposed AST structure.

Next, to investigate the source code complexity of the symbol generation structure, the cyclomatic complexity is calculated for the symbol table generation structure and the analysis structure for obtaining symbol information using SourceMonitor which is a complexity calculation tool. McCabe suggested cyclomatic complexity. If the complexity exceeds a certain value, it is difficult to manage the code. Therefore, it is recommended to keep the complexity below 10, and Microsoft's MSDN recommends keeping the complexity below 25. Table 1 shows the bad fix probability according to the cyclomatic complexity[3]. The bad fix probability represents the odds of introducing an error while maintaining code.

Table 1. Bad Fix Probability According to the Cyclomatic Complexity

Cyclomatic Complexity	Bad Fix Probability
1 ~ 10	5%
20 - 30	20% - 40%
50-100	40%
> 100	60%

Next, Figure 12 shows the result of analyzing the source code complexity of the existing symbol generation structure and the proposed symbol generation structure. The existing symbol generation structure has a complexity of 27.5 on average, another error of 20-40%, and a probability of error of 60% in routines with a maximum of 405 complexity. The average complexity of the proposed symbol generation structure is 1.94, and the error probability is less than 5%. The maximum complexity is 6, so that the probability of error occurrence is also less than 5% for the most complex routines. The generated code has a block depth of up to 9 and a mean block depth of 4.77, whereas the proposed symbol generation structure has a maximum block depth of 4 and an average block depth of 1.53. So, it can be seen that the routine is simplified. As a result of

analysis, it is confirmed that the complexity and the block depth of the proposed generation structure are greatly improved compared with the existing generation structure.

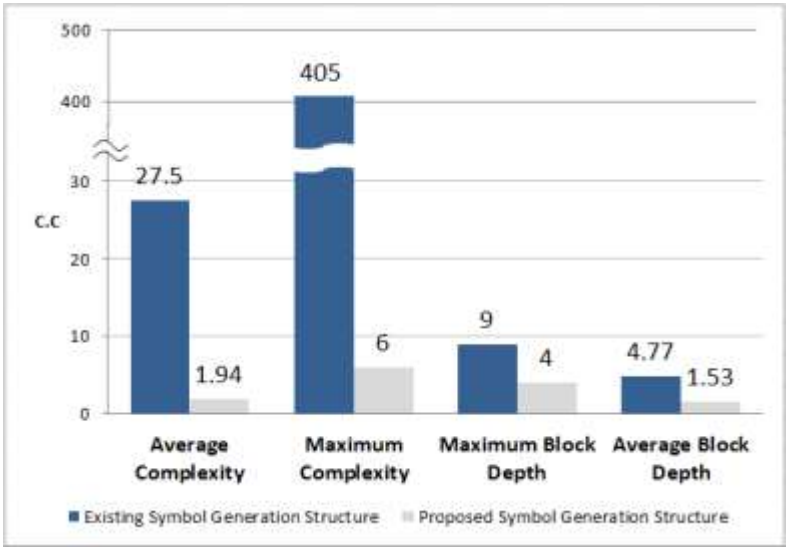

Figure 12. Source Code Complexity Analysis Result of Existing Symbol Generation Structure and Proposed Symbol Generation Structure

Figure 13 shows the results of the cyclomatic complexity analysis of the intermediate language generator module that generates the intermediate language by referring to the existing symbol table and the improved symbol table. As a result of investigation of the complexity of the routine for generating the intermediate language by referring to the symbol table, the intermediate language generation structure using the existing symbol table has a complexity of 11.5 on average and another error of 20-40%. Also, the routine with the maximum complexity has a value of 361 and has a probability of occurrence of error of 60%. In contrast, the intermediate language generation structure using the proposed symbol table exhibits an average complexity of 1.67, the probability of error occurrence is less than 5%, and the routine with the maximum complexity has a complexity of 33, which shows 90% improvement compared to the existing structure.

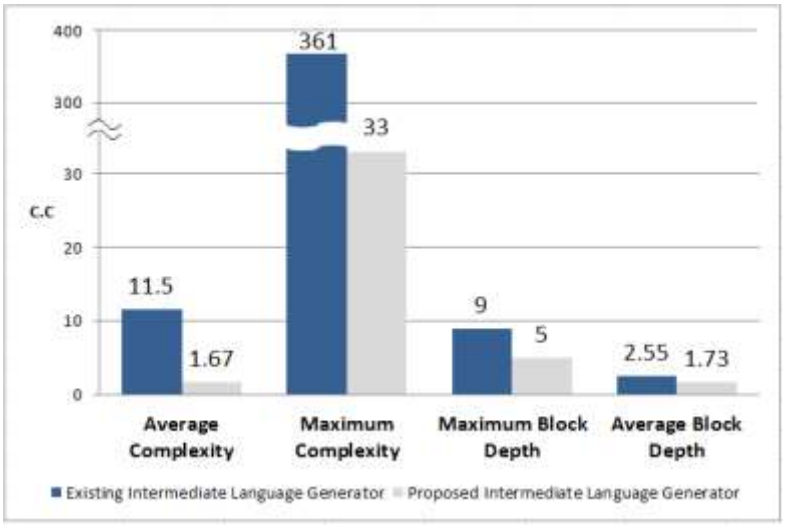

Figure 13. Source Code Complexity Analysis Result of the Intermediate Language Generation Structure using the Existing Symbol Table and the Proposed Symbol Table

The generated code has a block depth of up to 9 and a mean block depth of 2.55, while the proposed structure has a maximum block depth of 5 and an average block depth of 1.73. It can be seen that the routine for generating the intermediate language is simplified by referring to the symbols. As a result of the analysis, it is confirmed that the complexity and the block depth of the intermediate language generation structure using the proposed symbol table are greatly improved when using the existing symbol table. As a result of the analysis, it can be seen that the average and maximum complexity are improved when the existing symbol table and the proposed symbol table are used in both the symbol generation structure and the intermediate language generation structure.

5. Conclusions

Since a conventional symbol table can't represent symbols in one table, symbol attributes are divided and stored in a plurality of tables. A module referring to a symbol table refers to each of a plurality of tables, collects attributes, and had to complete symbol information again. In addition, in the case of an abstract syntax tree used to express the characteristics of a language and to generate symbols, since the language features are expressed through simple nodes, the abstract syntax tree has to be newly designed to add linguistic features even if the basic object-oriented language is the same, and new symbol generation routines and symbol table reference routines for the languages have to be added. Such an abstract syntax tree structure and a symbol expression scheme increase the complexity of a routine for generating symbols and a routine for referring to symbols.

In order to solve this problem, this paper has designed hierarchical structure of AST nodes by applying object orientation principle and Visitor pattern and improved the symbol table by hierarchical structure of symbols. With the improved AST structure, the symbol generation routine analyzes the nodes of the AST without additional conditional statements, and collects symbol information and makes it possible to easily add additional linguistic features into the hierarchical subclass form. The complexity of the symbol generation routine can be reduced because the symbols are assembled into the symbol table by the node traversal and inserted into the symbol table using the object orientation principle. Since the improved symbol table stores symbols designed in a hierarchical structure, symbol information can be directly obtained from the symbol reference routines. Therefore, a module using a symbol table simplifies the process of obtaining symbol information, and even if the information of the symbol is complicated, the additional complexity is not needed, and the complexity becomes simpler than the conventional one.

Acknowledgement

This Research was supported by Seokyeong University in 2015.

References

[1] S. C. Dewhurst, "Flexible symbol table structures for compiling C++," Software - Practice and Experience, vol. 17, (**1987**), pp. 503-512.
[2] R. P. Cook, T. J. Leblanc, "Symbol table abstraction to implement languages with Explicit Scope Control," IEEE Transactions on Software Engineering, vol. 9, (**1983**), p. 8.
[3] S. C. Dewhurst, "Flexible symbol table structures for compiling C++," Software: Practice and Experience, vol. 17, no. (**1987**), pp. 503-512.
[4] M. Gallego-Carrillo, F. Gortázar-Bellas, J. Urquiza-Fuentes and J.Á. VelázquezIturbide, "SOTA: A visualization tool for symbol tables," ACM SIGCSE Bulletin, vol. 37, (**2005**), p. 385
[5] J. F. Power, B. A. Malloy, "Symbol Table Construction and Name Lookup in ISO C++," 37th International Conference on Technology of Object-Oriented Languages and Systems, TOOLS 2000, (**2000**), p. 57.
[6] Y. S. Son, Y. S. Lee, "The Reverse Translator for Symbol Table Verification in Objective C compiler on Smart Cross Platform," The Asian International Journal of Life Sciences, vol. 11, (**2015**), p. 625.
[7] Y. S. Lee, Y. S. Son, "The Reverse Translator for Reconstruction of the Declaration Part from Symbol Tables of the C++ Compiler," Information-an International Interdisciplinary Journal, vol. 16, (**2013**), p. 2319.

[8] Y. S. Lee, Y. S. Son, "A Study on Verification and Analysis of Symbol Tables for Development of the C++ Compiler," International Journal of Multimedia and Ubiquitous Engineering, SERSC, vol. 7, (**2012**), p. 175.
[9] Y. S. Lee, Y. K. Kim, H. J. Kwon, "Design and Implementation of the Decompiler for Virtual Machine Code of the C++ Compiler in the Ubiquitous Game Platform," LNCS, vol. 4413, (**2007**), p. 511.
[10] H. J. Kwon, Y. K. Kim, J. K. Park, Y. S. Lee, "Design and Implementation of a Detranslator for Verification and Analysis of Symbol Tables in an ANSI C Compiler," The 2006 International Conference on Multimedia, Information Technology and its Applications (MITA2006), (**2006**), Dalian, China.
[11] Y.S. Lee, "The Virtual Machine Technology for Embedded Systems," Journal of the Korea Multimedia Society, vol. 6, (**2002**), p. 36.
[12] R. C. Martine, Agile Principles, Patterns, and Practices in C#, Pearson Education, (**2006**).
[13] E. Freeman, Elisabeth Freeman, K. Sierra, B. Bates, Head First Design Patterns, O'Reilly, (**2004**).
[14] L. M. Laird, M. C. Brennan, Software Measurement and Estimation: A Practical Approach, WILEY, (**2006**).
[15] M. H. Halstead,, Elements of Software Science, New York: Elsevier North-Holland, (**1977**).
[16] SourceMonitor, http://www.campwoodsw.com/sourcemonitor.html

Author

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 2004-2005, 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 2014-2016. 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.