Implemented Analyzer and Syntesis in Compiler Process

Tri Ichsan Saputra
Universitas Nasional
Jakarta, Indonesia

Fauziah
Universitas Nasional
Jakarta, Indonesia

Andez Apriansyah
Universitas Nasional
Jakarta, Indonesia

Gatot Soepriyono
Universitas Nasional
Jakarta, Indonesia

ABSTRACT
The compiler is a translator that is used in the process of making program code with the aim of tracing errors that occur. In this study, the compiler will function as a translator and as an error identifier. To make translators like compilers, standards or rules (grammar) need to be made, just as humans communicate to have grammar so that their interlocutors understand what is being discussed. The compiler works based on its structure, the system flow in the compiler is divided into 2 main groups, namely analysis and synthesis groups. In this study the analysis is in the form of lexical, syntactic and semantic analysis, also produced a synthesis of intermediate code, code generator, also code optimization. Later the program code will display an error message if there is an error in the program code that has been entered.

General Terms
Language Concepts and Notations: Grammar is a set of variables, terminal symbols, non-terminal symbols, initial symbols that are limited by production rules. Making a translator in the form of a compiler is needed grammar, as humans communicate having grammar so that the interlocutor understands. Likewise to translate into a machine (computer) must be made a grammar so that the computer can understand what is desired by humans through the programs they make [1].

Compiler: A program that can read programs that are source languages and translate them into machine language or the target language. The compiler maps program resources into the target program semantically. If seen, there are two parts of this compiler mapping: analysis and synthesis [2].

Keywords
Compiler, Language Notation, Analysis, Synthesis

1. INTRODUCTION
A compiler is system software that converts a high-level programming language program into a target language equivalent to low-level (machine) language program [3].

Besides that, the compiler can identify an error in the program code that the programmer enters so that it can be handled easily. Both functions of the compiler are needed, either as an interpreter or as an error identifier.

But every time we make a source code and start the evaluation process, the computer only displays output and errors (if they occur). We as programmers don't know the real process behind it. In this research paper, the exact procedure behind the compilation task is explained. Thus programmers can better understand the procedures that have been experienced [4].

2. RESEARCH METHODS
2.1 Compiler Structure

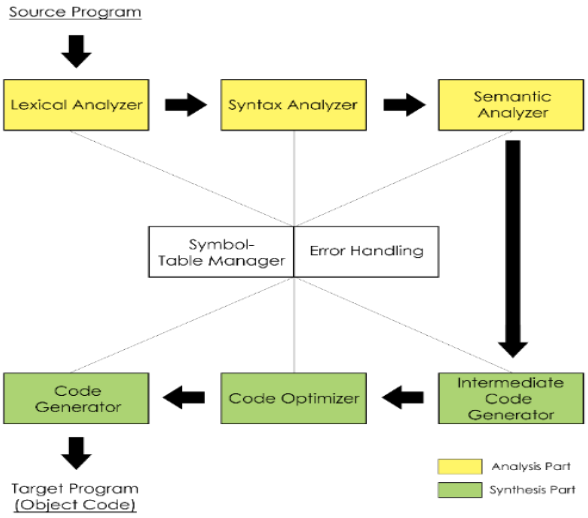

Fig 1: Structure of compiler

The compiler works based on its structure, Figure 1 describes the flow of the application design system which is divided into 2 phases, namely the analysis and synthesis phase.

2.1.1 Phase Analysis

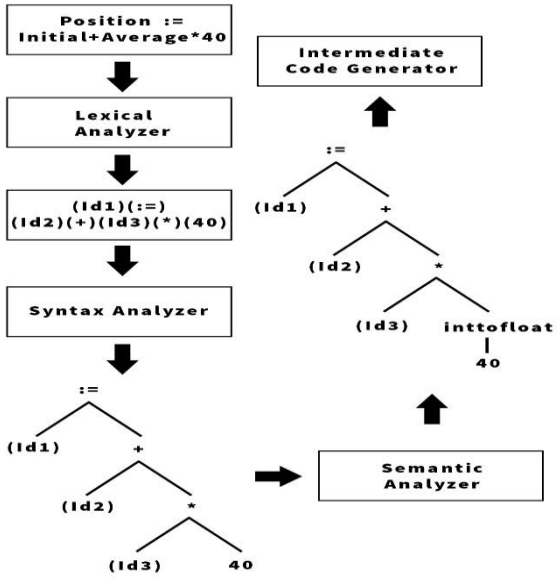

Fig 2: The design of the process that occurs in the analysis phase

1. Lexical Analyzer is the initial steps carried out in the compiler, also called scanners. The main process carried out in Lexical Analyzer is to convert an input sentence line into a token and then group it into lexemes, whitespace removal, conversion of numerical constants into specific/specific data types [5]. Then, the token will be sent to the next stage, namely Syntax Analysis.
2. Syntax Analyzer is also called parser analysis because the process carried out in this stage is the result of lexical analysis (token) will be arranged and grouped in a certain structure that has a specific definition [6]. The stages of the syntax analyzer are (1) sorting of tokens which are the results of the lexical analysis. (2) Continue by calling the next process, namely Semantic Analyzer.
3. Semantic Analyzer, this stage becomes a bridge between analysis and synthesis of a compilation. The final result of this step is an executable code in a simple compilation which is then manipulated with various optimization from the translator [2].

2.1.2 Phase Synthesis

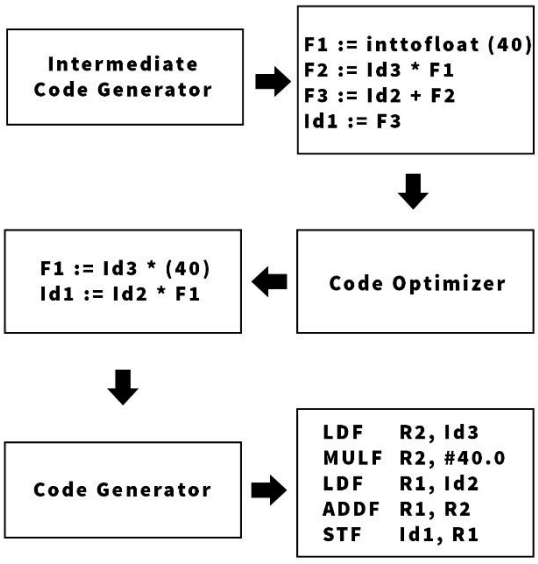

Fig 3: The design of phase synthesis

1. Intermediate Code Generator, the process for generating code based on the parsing tree (according to the related formula/syntax). The result of this process is a command that is a 3-address-code or quadruples [7].
2. Code Optimizer, the process for optimizing code that has undergone processes before finally forming an executable code. In this stage, one of them is the process of reducing redundancy in the code so that it is more efficient and effective [8].
3. Code Generator, the process of making code known as machine language which is an assembler language. Usually, the code consists of a command with an address and accumulator, each of which consists of 1 component [8].

3. RESULTS AND DISCUSSION

3.1 Lexical (Scanner)

Lexical Analysis (Scanner) checks the source code with status changes that occur. In its application, source code will be broken down into certain symbols called tokens [9]. The token specifications needed are terminal components contained in grammar. Scanner acts as an interface between the source code and the process of syntactic analysis (parser) [10]. The scanner will check each character from the source code. Lexical analysis deals with writing programming languages, for example, endl, float, string.

3.2 Syntaxis (Parser)

The tree is a connected graph that is not circular, has one node (or vertex), namely the root and from this root has a path (or edge) to each other node [11]. Derivation tree/syntax tree/parse tree is useful to describe how to get a string (string) by lowering or replacing variable symbols into terminals. Each variable symbol will be lowered or replaced into a terminal [4].

- The variable symbol is denoted by uppercase (capital).
- The terminal symbol is denoted by lowercase letters, occupying the position of the leaf.
- The initial symbol is the variable S, occupying the top of the tree (root).

The process of declining (or parsing) can be done, among others by:

a. Decrease through the leftmost derivation: the extended leftmost variable symbol expanded first.

b. Decrease through rightmost derivation: the extended rightmost variable symbol that is expanded first.

As an example,

It is known that context-free grammar (CFG) has the following production rules [7]:

S ➔ aAS | a

A ➔ SbA | ba

The following is a picture of the declining tree to get the strand 'aabbaa':

a. By means of leftmost decline, it will be obtained:

S >> aAS >> aSbAS >> aabAS >> aabbaS >> aabbaa

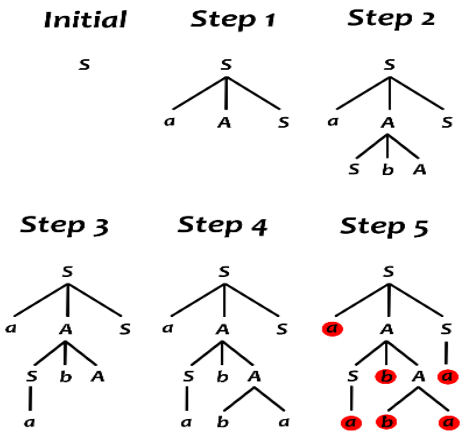

Fig 4: Leftmost derivation

b. By means of rightmost decline, it will be obtained::

S >> aAS >> aAa >> aSbAa >> aSbbaa >> aabbaa

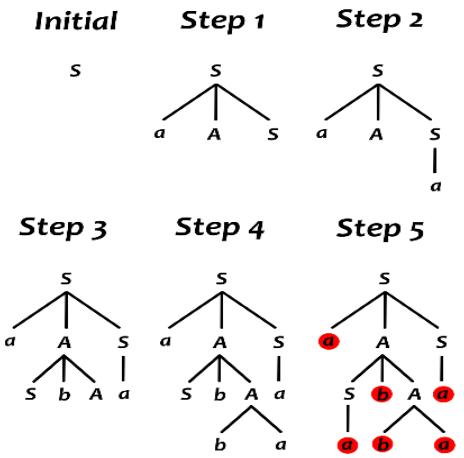

Fig 5: Rightmost derivation

3.3 Semantic

The semantic analysis utilizes the syntax tree generated in the parsing process. In general, the function of semantic analysis is to determine the meaning of a series of instructions contained in the source program [5]. Semantic analysis is related to variables in the program code, such as variables for name, total, results, etc.

3.4 Intermediate Code

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis [5].

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine [5].

In processing intermediate code, N-Tuple notation is used by quadruples notation.

The format for quadruples notation:

<operator> <operand> <operand> <result>

Example instruction :

Y := (A - B) * (C + D)

Then the form of quadruples is as follows :

a. - , A , B , T1

b. + , C , D , T2

c. * , T1 , T2 , Y

d.

3.5 Code Generator

The code generator is enabled to translate the code into assembly language or machine language [4].

Example : (A + B) / C * D

Quadruples :

a. + , A , B , T1

b. / , T1 , C , T2

c. * , T2 , D , T3

Assembly language translation (before optimization):

LDF A

ADDF B

STF T1

LDF T1

DIVF C

STF T2

LDF T2

MULF D

STF T3

3.6 Code Optimization

Code optimization is used to improve performance in terms of speed. Code optimization is carried out in an intermediate code so that repetition does not need to happen someday [13].

Assembly code in the above translation can be optimized to:

LDF A

SUBF B

STF T1

LDF T1

DIVF C

MULF D

STF T2

3.7 Error Handling

On CO:AS machines, the compiler will handle errors that occur in the source program, the form of handling the error is a message error. The error message will display a form of error from the analysis process, including lexical, syntactic and semantic analysis.

The following is part of the program code in handling errors:

```
string inputcode = tb_code.Text;

newwords.Clear();

onecheck.Clear();

string result = "";

bool error = false;

string temp = "";

for (int i = 0; i < inputcode.Length; i+

{

if (char.IsWhiteSpace(inputcode[i]))

{

i++;

while (i < inputcode.Length)
```

```
{
if (char.IsWhiteSpace(inputcode[i]))
{
i++;
}
else { i--; break; }
}
if (temp != "")
{
newwords.Add(temp);
}
temp = "";
}
else { temp += inputcode[i]; }
}
if (temp != "") newwords.Add(temp);
for (int i = 0; i < newwords.Count; i++)
{
if (_syntax(newwords[i]))
{
onecheck.Add(newwords[i] + " \t: Syntax Error" + System.Environment.NewLine);
}
else if (_leksikal(newwords[i]))
{
onecheck.Add(newwords[i] + " \t: Lexical Error" + System.Environment.NewLine);
}
else if (_semantic(newwords[i]))
{
onecheck.Add(newwords[i] + " \t: Semantic Error" + System.Environment.NewLine);
}
}
if (error) { }
else
{
for (int i = 0; i < onecheck.Count; i++)
{
result += onecheck[i] + "";
}}
frm2.SendToReceiver(result);
```

3.8 Software Design

For the design of the user interface of the CO: AS software (Compiler: Analyzer Synthesis) this uses 2 forms. The first form is the main page in which there is a text-box as the input source program that you want to compile and there are 3 buttons including the start button, clear button, and exit button can be seen from Figure 6.

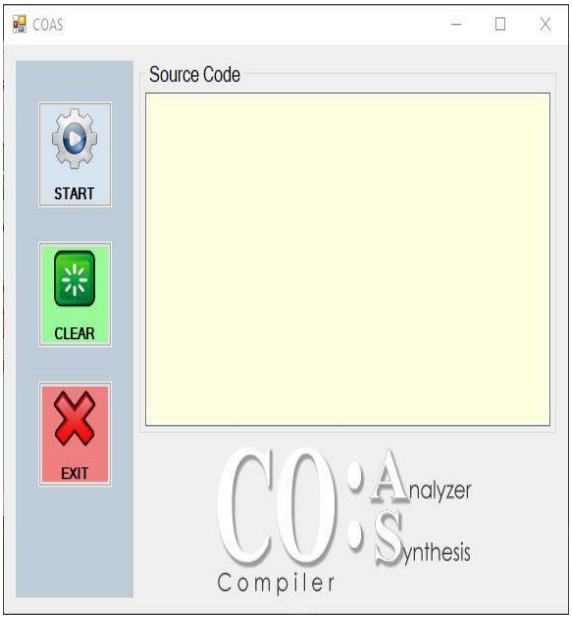

Fig 6: The main form of the program

The function of the start button is to execute the program code entered by the user into the text-box source code. The appearance after the user enters the program code can be seen in Figure 7.

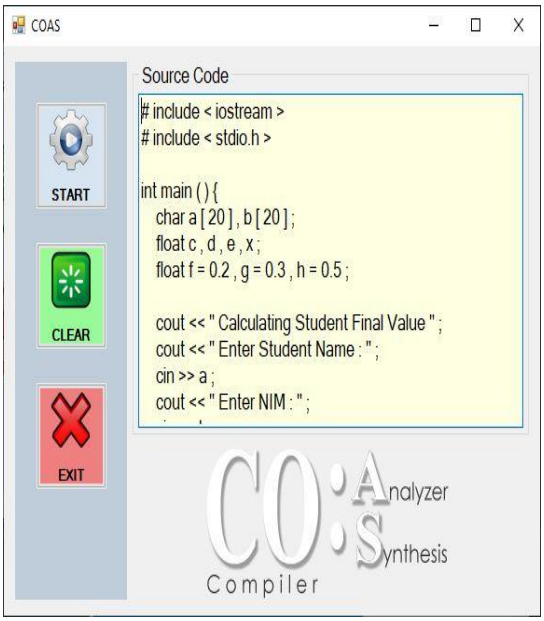

Fig 7: Input process

After the user enters the program code and presses the start button, the application will analyze the program code and display it on the second form. If there is no error in the program code, the application will not display an error message and will display the code which can be seen in Figure 8.

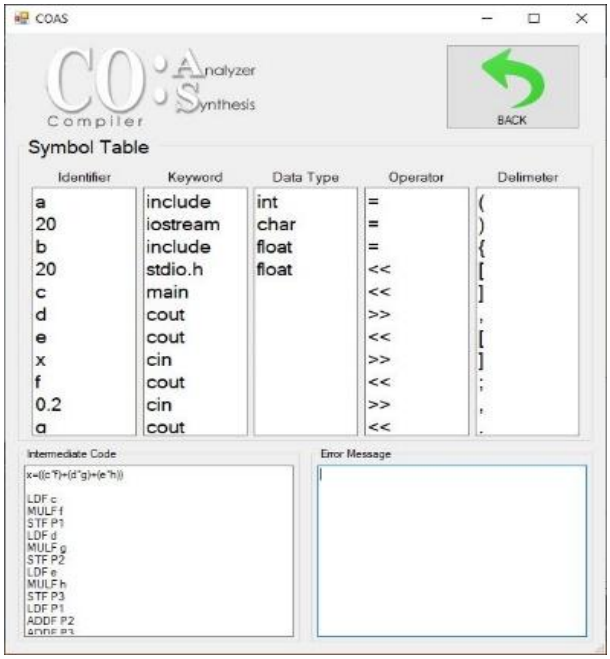

Fig 8: Successful in program execution

On the contrary, if the program code has an error whether it's a lexical error, the syntax or the semantic, it looks like in Figure 9.

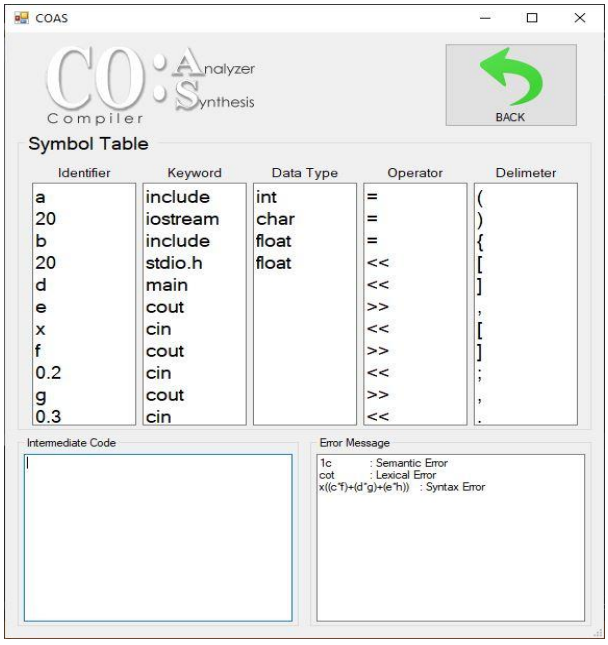

Fig 9: There was an error message on the program execution

4. CONCLUSION

CO:AS (Compiler: Analyzer Synthesis) software created and designed to run compilation functions properly. The results of this study are symbol tables, message errors in terms of lexical analysis, syntax and also semantics, this study also produces optimization results from intermediate code.

5. REFERENCES

[1] M. Jain, N. Sehrawat, and N. Munsi, "Compiler Basic Design and Contruction," Int. J. Comput. Sci. Mob. Comput., vol. 3, no. 10, pp. 850–852, 2014.

[2] R. Babu, V. Tiwari, and J. Dehakar, "Parsing and Compiler design Techniques for Compiler Applications," Int. J. Recent Innov. Trends Comput. Commun., vol. 3, no. 2, pp. 449–453, 2015.

[3] P. Saini and R. Sharma, "An Analysis of Compiler Design in Context of Lexical Analyzer," Int. J. Emerg. Technol., vol. 8, no. 1, pp. 377–381, 2017.

[4] V. Trivedi, "Life Cycle of Source Program - Compiler Design," Int. J. Innov. Res. Comput. Commun. Eng., vol. 5, no. 12, 2017.

[5] P. Ezhilarasu and N. Krishnaraj, "Applications of Finite Automata in Lexical Analysis and as a Ticket Vending Machine – A Review," Int. J. Comput. Sci. Eng. Technol., vol. 6, no. 05, pp. 267–270, 2015.

[6] G. P. Arya, N. Sohail, P. Ranjan, P. Kumari, and S. Khatoon, "Design and Implementation of a Customized Compiler," Int. J. Comput. Sci. Inf. Technol., vol. 8, no. 3, pp. 342–346, 2017.

[7] L. Leiva and N. Acosta, "MISD Compiler for Feature Vector Computation in Serial Input Images," ARPN J. Syst. Softw., vol. 1, no. 3, pp. 108–116, 2011.

[8] P. P. Gotarane and S. N. Pundkar, "Java Compiler : Review on Code Generation and Optimization Techniques," Int. J. Innov. Adv. Comput. Sci., vol. 4, no. March, pp. 66–76, 2015.

[9] A. Goyal and R. Yadav, "Content and Lexical Analysis : Practical Application," Int. J. Comput. Sci. Mob. Comput., vol. 3, no. 10, pp. 356–366, 2014.

[10] R. Chuhan, V. Singh, K. Makhan, and M. Kumar, "Implementation of Lexical Analysis," Int. J. Res. Appl. Sci. Eng. Technol., vol. 5, no. V, pp. 614–617, 2017.

[11] T. Tyagi, A. Saxena, S. Nishad, and B. Tiwari, "Lexical and Parser tool for CBOOP program," IOSR J. Comput. Eng., vol. 10, no. 6, pp. 30–34, 2013.