Parsing and Evaluation of Mathematical Expresion with Regular Expression, Recursive Descent, and Divide and Conquer Algorithm

Ridho Pratama — 13516032[1]
Program Studi Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132,Indonesia
[1]13516032@std.stei.itb.ac.id

***Abstract*—We use mathematical expressions in our daily life, but we usually just type the expression, press enter, and then we will get the results, and the same thing with program codes. The compiler is able to turn the input which is a sequences of characters, make sense of it, and make results out of it. In this paper, author will talk about the steps taken by basic modern compiler to be able to do that, and the supporting theories and algorithm used like formal grammar, context-free grammar, EBNF, AST, Regex, recursive descent, and Divide and Conquer.**

***Keywords*—Divide and Conquer, Lexing, Parsing, Recursive Descent, Regular Expression.**

I. Introduction

In our daily life as a programmer, or at least as someone who write codes and execute it, we usually just write it in out favourite text editor, compile, and run the result without thinking how the compiler, or interpreter, able to execute the program. How it able to make sense of the codes that the computer sees as sequences of bytes.

Nowadays, a programming language compiler or interpreter usually do three steps. Lexing, parsing, and evaluation. Lexing takes the sequences of bytes and turns them into sequences of token. Parsing takes the sequences of token and then generate abstract syntax trees. And evaluation evaluates the program and the results.[1] The kind of evaluation that happens usually differs by language and the language type. In compiled language like C, the evaluation step means evaluating the syntax tree and generating the binary executable for the target machine, which later is executed by the machine. In the other hand, an interpreted language like Python, the evaluation step is generating bytecodes, and the executing the bytecodes with the language virtual machine.

In this paper, we will look at how to parse and evaluate an mathematical expression like `5 + 1 / 2 ^ 3`, with regular expression for lexing, recursive descent for parsing, and Tree-Walking Evaluation for evaluating the expression. While it may look simple, these techniques could be expanded and implemented further to make a simple programming language interpreter, but with drawbacks like slow runtime.

II. Theory

A. Formal Language and Grammar

1) Formal Language: In the field of mathematics, computer science, and linguistic formal language is a set of strings of symbols and a set of rules specific to it. These rules define if a sequences of letters, symbols, or tokens are in that formal language. These rules are usually called formal grammar.

Formal language is first studied as an attempt to describe the grammar of natural human language. But natural human language is too complex and full of exception that it is impossible to describe a formal language that will fit exactly to natural human language.

Formal language founds another use in programming language where a certain type of formal language called context-free language could be used to describe programming language, which must have a solid description and have no ambiguity so no errors or bugs could happen.

At first, after defining an grammar for a programming language, programmers usually code the parser by hand. But as the study of formal language for programming language continues, computer scientist were able to found a way to create a program that could, given a grammar, create a parser program for that grammar, which really help.

2) Context-Free Grammar: Context-free grammar is a certain type of formal grammar where all the rules are just simple replacement. An example of a context-free grammar is: Where the Greek letters are terminals, and the uppercase letters

$$\begin{aligned} A &\rightarrow \alpha \mid \beta \\ B &\rightarrow A \mid A\ B \\ C &\rightarrow \gamma \end{aligned}$$

Figure 1: An example of a context-free grammar

are non-terminals. In the example grammar, “B –> A | A B” means that a non-terminal “B” could be replaced with “A” or “A B”.

A set of strings of symbols must only consist of terminals and could be produced from a starting symbol with the available rules. In the example grammar, with the starting symbol “B”, string “A B” is not a part of the grammar because it contains

non-terminal, string "gamma" is not in the grammar because there is no production from "B" that could produce "gamma", and "alpha" is part of the grammar because there is production that could produce that.[2]

One special point of context-free grammar is that it is context-free and have no memory, so it only have the rules to guide the production, and no other external influences.

Context-free grammar is the type of formal grammar used to express the grammar used in programming languages.

3) Extended Backus-Naur Form: Extended Backus-Naur Form or EBNF, is an alternative notation of expressing context-free grammar, because the notation used in figure II-A2 is rooted in the mathematical and linguistic field of formal grammar. To make it more practical, in 1959, John Backus and Peter Naur developed what Donald Knuth would call "Backus-Naur Form" which is first used to describe syntax of ALGOL 58.

Later Niklaus Wirth makes some extension to the Backus-Naur Form and later ISO adopted Wirth's works as Extended Backus-Naur Form (ISO/IEC 14977).

The grammar in 1 could be described in EBNF as:

$\langle A \rangle$ ::= α | β ;
$\langle B \rangle$ ::= $\langle A \rangle$ | $\langle A \rangle$ $\langle B \rangle$;
$\langle C \rangle$ ::= γ ;

Figure 2: Grammar in figure 1 described in EBNF

EBNF also have additional syntax to express more things like repetition, grouping, exception, and optional parts.

In this paper, we will use EBNF to describe the grammar for mathematical expression.

B. Lexing

Lexing in the process where a source code like "1 + 2" is turned int sequence of tokens like

```
[number] 1
[plus] +
[number] 2
[EOF]
```

Figure 3: Example of tokens

While we could stuff the tokens definition into the language grammar, it is more simple of we separate the definition of tokens and the language grammar because then the grammar definition could be freed from the definition of the language tokens.[1]

A token is defined as a structure that, at the very basic level, have two fields, `type` defines the type of the token, like `number`, `plus`, or `EOF` etc. Then `literal` is the slice of the code that corresponds to that token. And in more advance lexer, the tokens also contains the position of the token to help in error reporting.

In figure 3, the first token, `[number] 1`, have `number` as the `type`, and `1` as the `literal`.

In reality, writing a lexer by hand is not that hard, but when the definition of a token changes, we have to be very careful in changing the corresponding code. This should not happen very often because in programming language design, we usually expects that definition of token to not change, but only expanded to accommodate new token definition.

Other than writing a lexer by hand and manually code the rules by hand, there is another way of creating a lexer, that is by utilizing another part of formal language theory, called regular expression.

1) Regular Expression: Regular Expression is a sequence of characters that define a patterns. This patterns will then be used to find matches in strings, or for string validation. Regular Expression is created to suit matching ASCII, Unicode, or the kind of string we usually sees when reading a text, unlike EBNF which work for a higher and abstract definition of alphabet and token.

Regular Expression in formal language, is the rules that defines a regular language.

While EBNF tries to match sequences of tokens to grammar and make trees out of it, Regex tries to find matches in string that match the defined pattern.

A few examples of regex patterns are:

- `a` matches to string "a",
- `ab` matches to string "ab",
- `a*` matches to zero or more "a",
- `a+` matches to one or more "a",
- `a?` matches to zero or one "a",
- `a|b` matches to "a" "b",
- `^a` matches to one "a" at the start of the string.

Usually regex matches only show the start and end position of the matching string. But using parentheses "`()`" we also could have the thing that we are matching for in the parentheses to be matched too, and we could have multiple and nesting parentheses to get multiple part of the results. This is known as capture groups. For example, `(a*)` will return the position of matching string, and also the contents of the parentheses.

As we could see, the regex system is really flexible and really adequate for our need in lexing, that is to match part of string and get the results of that match.

2) Utilizing Regular Expression in Lexing: To use regex in our lexer, first we have to list the tokens that we needed for mathematical expression. An example of a mathematical expression that use all of the available features is `(1 + 2) ^ 5 / 3 * 4 - 6`. From that example, we could see that the tokens that we use are `lparen`, `rparen`, `caret`, `number`, `slash`, `plus`, `star`, `minus`.

To make it more clear, the lexer result for our examples is listed in figure 4.

```
[lparen] (
[number] 1
[plus] +
[number] 2
[rparen] )
[caret] ^
[number] 5
[slash] /
[number] 3
[star] *
[number] 4
[minus] -
[number] 6
[EOF]
```

Figure 4: Lexer result for `(1 + 2) ^ 5 / 3 * 4 - 6`

We will refer this result of tokens for later examples.

The tokens are pretty self-explanatory, with the addition of "EOF" to marks the end of the tokens.

After we know what are the tokens that will be use, we have to write the definition in regex. As for the operators which are single character long, we could use regex `(\+)` to match for +, the \is used because + is a special character in regex, so we have to use the backslash to escape it. We will call the regex patterns that we defined for our tokens type as rules. We will store this rules as an dictionary, where the keys is the rule name, and the value is the regex pattern.

After we defined the rules, we will do our lexing according to this pseudocode:

```
results ← empty list of tokens
rules ← defined dictionary of regex rules
input ← source code
while input is not empty
  find the rule that have the longest
  match and starts at the start of
  input
  if have match:
    add the corresponding token to results
    remove the matching part from input
  else:
    throw error that no rules match
return results
```

Figure 5: Pseudocode for our lexer

We first search rules that will have a match at the start of the string. We could just use ^ in all of our rules definition, but we know that we always try to match at the start of the string, so rather than adding ^, we could takes the matches that starts at 0 (or 1 if our language 1-based). From those matches that match our criteria, we will take the longest match. It is because there might be some ambiguities, like if we defined a token `for` to be used in for-loops and reserved, and token `identifier` for any other keywords that is not in the reserved keywords, when we get input `foreign`, if we just take the first rule that match, we will get `[for] for` and `[keyword] ign` while what we wanted is `[keyword] foreign`. That is why when we get multiple matches, we should take the longest match. If after that we still have multiple matches, then the rules should be revised so that there are no multiple possible result for a string.

So the regex rules for our tokens are

Token Type	Regex
Number	`(\d+(\.\d+)?)`
Plus	`(\+)`
Minus	`(\-)`
Star	`(\*)`
Slash	`(/)`
Caret	`(\^)`
Lparen	`(\()`
Rparen	`(\))`

Figure 6: Tokens and their regex definition

C. Parsing

After lexing, the next step is parsing. Parsing is the process of taking a sequences of tokens and with a set of grammars turns them into an abstract syntax trees that have meanings and could be parsed by computer. The trees that we talk about is the same kind of the trees in graph theory, which have a node, and zero or more children.

1) Abstract Syntax Tree: Abstract Syntax Tree (AST) is a tree that represents the abstract syntactic nature of a source code. Each node of the tree represents a structure in the programs. An example is an if-else condition could be a tree with three children, first is the condition, second is the if part, and third is the else part. The children could be another tree, or just a single value of number, expression, or statements.

To express mathematical expression, we need three kinds of AST. First is `Value`, it represents a single number or value like `9`, so it only has one child, which is the number itself. Second is `Binary`, that represents binary operation between two expressions like `1 + 2`, so it has three children, the left expression, operator, and the right expression, where operator is only storing the token that used for operator. And the last is `Grouping`, that the represents the grouping when we used parentheses to force precedence. It only has one child, that is the expression inside the parentheses.

We could express the trees with EBNF

⟨*Expression*⟩ ::= ⟨*Binary*⟩ | ⟨*Grouping*⟩ | ⟨*Value*⟩ ;
⟨*Binary*⟩ ::= ⟨*Expression*⟩ operator ⟨*Expression*⟩ ;
⟨*Grouping*⟩ ::= lparen ⟨*Expression*⟩ rparen ;
⟨*Value*⟩ ::= number ;

Figure 7: Basic AST of math expression

Where ⟨*Expression*⟩ acts as our starting symbol.

Then we could express `(1 + 2) ^ 5 / 3 - 4 * 6` (I've changed the operators to make it more interesting) in AST as:

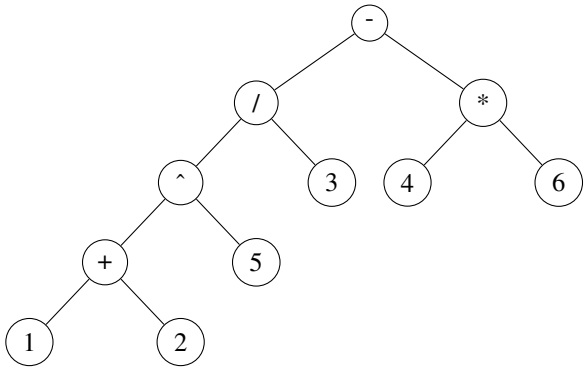

Figure 8: AST representation of `(1 + 2) ^ 5 / 3 - 4 + 6`

And the AST in figure 8 also applies the correct precedence. But our grammar in figure 7 doesn't have any information about operator precedence, so we have to fix that. While for AST we will use the grammar in figure 7, we will modify that we could easily translate it to recursive descent parser, and also able to force precedence.

There is other way of parsing mathematical expression like Shunting-Yard Algorithm, but we will not talk about it, since in only applies to mathematical expression alone, and won't applies to general programming language parsing.

2) Operator Precedence: While the AST in figure 7 could represent a mathematical expression, it is still full of ambiguity since our grammar could not force operator precedence.

To solve this problem first we look at operator precedence table

Name	Operators
Unary	-
Exponent	^
Multiplication	/ *
Addition	+ -

Figure 9: Operator precedence table

Where the first row is the highest precedence, while the last row has the lowest precedence.

What the table in figure 9 tells us is when we got an expression with mixed precedence, we should prioritize the one with higher precedence. So from our binary grammar which is *<Binary> ::= <Expression> op <Expression> ;*, we could make a special grammar for multiplication like *<Multiplication> ::= <Exponent> ('/' | '*') <Exponent> ;*, where we use unary to to tell that our grammar should prioritize exponent over multiplication and that we only accept slash or star as the operator.

Then we would like for our grammar to be able to express chains of binary expression like `1 + 2 + 3` but our grammar only seems to be able to express binary expression with two number. We could use repetition to repeat the operator and right expression. So our multiplication grammar turns into *<Multiply> := <Exponent> (('/' | '*') <Exponent>)*;*, which means that a multiplication expression is a unary expression followed by zero or more *('/' | '*') <Unary>* sequences. So we do this process to our grammar, incorporating all of the operator precedence that results in:

⟨*Expression*⟩ ::= ⟨*Addition*⟩ ;
⟨*Addition*⟩ ::= ⟨*Multiply*⟩ ((plus | minus) ⟨*Multiply*⟩)* ;
⟨*Multiply*⟩ ::= ⟨*Exponent*⟩ ((star | slash) ⟨*Exponent*⟩)* ;
⟨*Exponent*⟩ ::= ⟨*Unary*⟩ (caret ⟨*Exponent*⟩)? ;
⟨*Unary*⟩ ::= (minus ⟨*Unary*⟩) | ⟨*Primary*⟩ ;
⟨*Primary*⟩ ::= number | ⟨*Grouping*⟩ ;
⟨*Grouping*⟩ ::= lparen ⟨*Expression*⟩ rparen ;

Figure 10: Grammar with precedence, adapted from [3]

There might be other way to write the grammar, but writing it this way makes it easier to be adapted to the parsing algorithm that we will use, that is recursive descent.

3) Recursive Descent Parsing: There is a lot of algorithm that could be used for parsing, like LALR, which used by Yacc or GNU Bison. The advantages of using LALR are the limit of the grammar that we could use is less than using recursive descent parser, and the aforementioned Yacc and GNU Bison is a "compiler-compiler", a program that creates a compiler. But the disadvantages are it is complex, we can't really control what happen in the parsing, and by using finished tools we can't really learn things.

By using recursive descent, we could understand how a parser works, able to tune the parser with our hands. Also, by conforming our grammar to some rule (which called LL(k) grammars), we could just turns the grammar into code with the same structure. The GNU C Compiler (GCC) at one time used Bison, but then later changed back to using recursive descent.[4]

The way recursive descent works, like the name implies, is by using recursion. Each rule in our grammar turns into functions, which the contents is like the rule. The function then returns the corresponding AST, like the `Addition`, `Multiply`, `Exponent` and `Unary` will return Binary AST, `Grouping` will return Grouping AST, and the `Primary` might return a Literal or Grouping AST.

One example of the function is:

```
def addition():
    expr = multiply()
    while next token is plus or minus:
        op = next token
        right = multiply
        expr = binary expression from
               expr, op, and right
    return expr
```

Figure 11: Example recursive descent code for addition rule

Now, we have the tools to generate an AST like in figure 8. The next step to do is to evaluate the AST and get the value.

D. Evaluation

For each kind of AST we have, we have to define how to evaluate it. Here we use the Divide and Conquer algorithm where we could separate the task of evaluating an expression into few smaller expression recursively.

1) Literal: Literal AST is an AST that only has one child, that is a value. So the evaluation of Literal will return the value of that child.

2) Grouping: Grouping AST is an AST that only has one child, the expression begin grouped. So evaluation of Grouping will evaluate the expression inside it, the returns the result.

3) Binary: Binary AST is an AST that has three children, the left expression, operator, and the right expression. First we evaluate the left expression, the we evaluate the right expression, and finally we combine the results using the operator that is has, and returns that result.

To note, we could express and unary grammar like `-5` as Binary AST of `0 - 5`. This is to simplify the amount of AST we use, but in real life implementation, it makes sense to have a separate unary operator since there might be other unary operator than `-` like `!` or `~`.

E. Miscellaneous

The way of doing an evaluation like described before is commonly called Tree-Walk Interpreter, because it is just walking the AST and evaluating the values to get the results. This method is easy to understand, but have drawbacks like slow evaluation time. But for this paper we will only use this method.

But there are some ways to make the evaluation step faster, but it will have it's own advantages and disadvantages, and also could affect the way the language is designed. These ways is usually used for real programming language, and don't really make sense when used for mathematical expression evaluation where the AST generated usually is not that large.

Also, we could also do some steps before evaluation, like resolving for variables, and do type checking. And in Rust Programming Language, the borrow checker runs before evaluation.

1) Generating Machine Code: Rather than blindly doing tree-walking, we could use the evaluation step to generate machine code, like x86 assembly, or the ARM code. The advantage of this method is that the program produced will be the fastest than other method, but the disadvantages are generating machine code (compiling) takes time, and for every machine architecture that a programming language, we have to write separate program to generate the machine code. The second disadvantages could be solved by using VM, or LLVM,

2) Virtual Machine: Rather than generating machine code, we could create a certain bytecode specification (like Java bytecode), and then write program that could run that bytecode (like JVM). This is the path taken by Java, Python, and many other languages, albeit Java being a bit different where the bytecode is saved as .class file which could be ran later, rather than Python that generate the bytecode, and the immediately run the bytecode that is saved in the runtime environment (look at Python's Language Service Library). The program generated by this method will be slower than generating machine code.

III. Implementation

The example of implementation will use Python.

A. Lexer

```
rules = {}  # type: Dict[str, str]
rules['number'] = r'(\d+(\.\d+)?)'
rules['plus'] = r'(\+)'
rules['minus'] = r'(\-)'
rules['star'] = r'(\*)'
rules['slash'] = r'(/)'
rules['caret'] = r'(\^)'
rules['lparen'] = r'(\()'
rules['rparen'] = r'(\))'


def Lex(to_match: str) -> List[Token]:
    to_match = to_match.strip()
    results = []  # type: List[Token]
    while len(to_match) > 0:
        max_rule = ''
        max_lit = ''
        to_match = to_match.strip()
        for rule in rules.keys():
            p = re.compile(rules[rule])
            m = p.match(to_match)
            if m is not None:
                if len(m.group(0)) > len(
                    max_lit):
                    max_lit = m.group(0)
                    max_rule = rule
        if max_rule == '':
            raise NoMatchException(
                to_match)
        to_match = to_match[len(max_lit)
            :]
        results.append(Token(max_lit,
            max_rule))
    results.append(Token('', 'EOF'))
    return results
```

B. Parser

```
class ParseError(Exception):
    def __init__(self, msg):
        self.msg = msg


class Parser:
    def __init__(self, tokens: List[Token
        ]) -> None:
        self.tokens = tokens
        self.current = 0

    def addition(self) -> expr.Binary:
        e = self.multip()
        while self.match(['plus', 'minus'
            ]):
```

```
            op = self.prev()
            right = self.multip()
            e = expr.Binary(e, op, right)
        return e

    def multip(self) -> expr.Binary:
        e = self.expon()
        while self.match(['star', 'slash'
            ]):
            op = self.prev()
            right = self.expon()
            e = expr.Binary(e, op, right)
        return e

    def expon(self) -> expr.Expr:
        left = self.unary()
        if self.match(['caret']):
            op = self.prev()
            right = self.expon()
            left = expr.Binary(left, op,
                right)
        return left

    def unary(self) -> expr.Expr:
        if self.match(['minus']):
            op = self.prev()
            right = self.unary()
            return expr.Binary(expr.
                Literal(0), op, right)
        return self.primary()

    def primary(self) -> expr.Expr:
        if self.match(['number']):
            try:
                return expr.Literal(float
                    (self.prev().literal))
            except:
                print("n {}".format(self.
                    prev()))
                exit()
        elif self.match(['lparen']):
            e = self.expression()
            self.consume('rparen', "
                Expected ')' after group
                expression.")
            return expr.Grouping(e)
        else:
            t = self.peek()
            raise ParseError("Syntax
                error: {}.".format(t))
```

C. Evaluation

```
class Calculator():
    def calculate(self, e: expr.Expr) ->
        float:
        if isinstance(e, expr.Literal):
            return e.value
        elif isinstance(e, expr.Grouping)
            :
            return self.calculate(e.expr)
        elif isinstance(e, expr.Binary):
            lval = self.calculate(e.left)
            rval = self.calculate(e.right
                )
            oprule = e.operator.rule
            if oprule == 'plus':
                return lval + rval
            elif oprule == 'minus':
                return lval - rval
            elif oprule == 'star':
                return lval * rval
            elif oprule == 'slash':
                return lval / rval
            elif oprule == 'caret':
                return lval ** rval
```

D. Main Code

```
print('> ', end='')
to_match = input().strip()
print("Input: {}\n".format(to_match))

try:
    results = lexer.Lex(to_match)
except lexer.NoMatchException as e:
    print("Error: {}".format(e))
    exit()

print('Printing tokens:')
for t in results:
    print(t)

print('')

printer = Printer()
c = Calculator()

print("Expresion Tree for: {}".format(
    to_match))
p = parser.Parser(results)
try:
    ex = p.parse()
except parser.ParseError as e:
    print("Parse error: {}".format(e))
    exit()

printer.print(ex)
print("Result: {}".format(c.calculate(ex)
    ))
```

IV. Experiment

```
> (1 + 2) ^ 5 / 3 - 4 * 6
Input: (1 + 2) ^ 5 / 3 - 4 * 6

Printing tokens:
[lparen] (
[number] 1
[plus] +
[number] 2
[rparen] )
[caret] ^
[number] 5
[slash] /
[number] 3
[minus] -
[number] 4
[star] *
[number] 6
[EOF]

Expresion Tree for: (1 + 2) ^ 5 / 3 - 4 * 6
-
├─/
│ ├─^
│ │ ├─+
│ │ │ ├─ 1.0
│ │ │ └─ 2.0
│ │ └─ 5.0
│ └─ 3.0
└─*
  ├─ 4.0
  └─ 6.0
Result: 57.0
```

V. Conclusion

In conclusion, to be able to turn source code into program, a compiler have to go through three steps.

First is Lexing, where the source code in the form of sequences of characters is turned into sequences of tokens that has the information of token type and the part of the string that corresponds to that token. To do this easily we could use Regular Expression so that we could write and makes change to the lexer easily.

Second is Parser, where the sequences of tokens is turned into AST which represents the abstract structure of the code. We could use EBNF to describe the syntax, and then convert the EBNF into recursive descent parser easily.

Third and the last is Evaluation, where we use divide and conquer to evaluate the children of the AST.

The methods described in this paper could be expanded to parse and evaluate a proper programming language.

Acknowledgment

First, author would like to thanks God for His blessing so that the author is able to write this paper.

Then, author would like to thanks his parents for supporting author.

Next, author wants to thanks Dr. Ir. Rinaldi Munir, MT., Dr. Nur Ulfa Maulidevi, ST. M.Sc, and Dr. Masayu Leylia Khodra, ST., MT for guiding author in IF2211 Strategi Algoritma course. Their willingness to teach and share their knowledges able the author to write this paper.

Finally, author also thanks author's friends for supporting author that enables author to write this paper.

References

[1] N. Chomsky, "Three models for the description of language." *IRE Transactions on information theory*, vol. 2, pp. 113–124, September 1956.

[2] S. Scheinberg, *Note on the Boolean Properties of Context-Free Languages*. Information and Control, 1960.

[3] B. Nystorm. (2017) Crafting interpreters. [Online]. Available: http://www.craftinginterpreters.com/

[4] (2006) Gcc 4.1 release series: Changes, new features, and fixes. Free Software Foundation. Accessed 25 April 2018. [Online]. Available: https://gcc.gnu.org/gcc-4.1/changes.html

[5] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Pearson, 2006.

Statement

With this I state that the paper that I wrote is my own writing, not an adaptation, or a translation of others papers, and is not plagiarized.

Bandung, 13th May, 2018

Ridho Pratama, 13516032