Recursive Descent Parser in Building A Simple Compiler

Manasye Shousen Bukit / 13516122
Informatics Undergraduate Program
School of Electrical Engineering and Informatics
Bandung Institute of Technology, Ganesha Street 10 Bandung 40132, Indonesia
13516122@std.stei.itb.ac.id
manasyebukit@gmail.com

***Abstract*—Recursivity is one of the most important concept in computer science and technology.One of its application is in building compiler.Compiler is a computer software that transforms one programming language,in example Pascal , C , C++; into another language that can be easily understood and processed by computer.The way programmer build a compiler may vary in method.One way to build a compiler is through a top-down parsing,recursive descent parser.**

***Keywords*—compiler, recursive , grammar , parsing**

I. Introduction

Humans can easily interact with others through media called language.There are many language that human can understand and learn.That is due to our brain capacity that can easily translate not only formal language,but also informal language and react towards it.On the other hand,computer does not work like that.Computer work in a specific way and can not interpret as flexible as the way humans do.

Regardless of the that disadvantages,computer do have an advantage.That is,computer process things a lot faster and a lot more neat than human.Let us take one simple example.We want to calculate how many prime numbers exist in range 0 - 100.If we count that manually,we would have a difficult calculation and tend to have error.That is when computer comes in handy.

In order to make computers understand what we want them to do,we need to make a programming language.You may be familiar with C,C++,Java,and so on.This is the example of high-level programming language.High-level language tend to adapt with human natural language in order to make humans easier to use it.

However,computer process things slight different than we expect and a lot more complex.We need some kind of a bridge that connect high-level programming language with computer instruction code.Therefore,compiler is built to solve this sort of problem.

Developing a compiler is not an easy thing.You need to understand grammar and its notation.Moreover,you need to develop a method that can easily parse through syntax in high-level language and give result as the user wants.There are several way to build a compiler.For instance, CYK Algorithm(Cocke-Younger-Kasami) a bottom-up syntax parser.There is also LR , LL Parser,and Recursive Descent Parser(RDP).[1]

Recursive Descent Parser is a way to apply a Context Formal Language(CFG) to make an analytic syntax processing in a certain code.Its distinguish characteristic is this method recursively derivate all variable until it meet the terminal with or without having a back-track. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.Another characteristic of this method is its dependency with scan algorithm to retrieve tokens.

II. RECURSIVITY, TREE, AND GRAMMAR

A.Recursivity

The process of defining an object in terms of itself is called recursion. A recursive function consists of two steps.

- Basis step : part of function that contain explicitly told value.This part is also the part that stop the recursive process.
- Recursive step : part of function where it call its own term while also getting closer towards basis. [2]

A function that does not have that two step is not a recursive function.For instance,a factorial function.We can define a recursive function with basis 0 that return value of function 1.The recursive step is (n-1)! with being multiplied with value of n.

$$n! = \begin{cases} 1 & ,n=0 \\ n\cdot(n-1)! & ,n>0 \end{cases}$$

The example of calculating 4 factorial using this recursive function is shown in picture below.

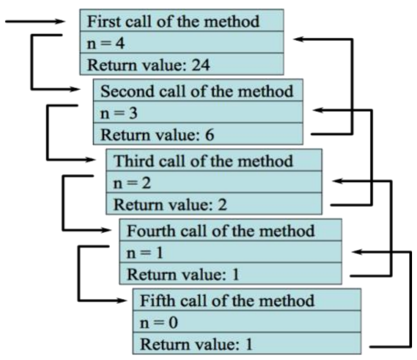


Figure 2.1 Determining factorial of 4 using recursive function (Source : https://i.stack.imgur.com/jqbf3.png)

Basis of a recursive function does not necessarily have to be just one condition.For example, Fibonacci sequence have two base.This Fibonacci sequence will have a result of sequence of 0,1,1,2,3,...

$$f_n = \begin{cases} 0 & ,n=0 \\ 1 & ,n=1 \\ f_{n-1}+f_{n-2} & ,n>1 \end{cases}$$

This recursive idea is widely used in many technology sector.In building a compiler , deriving a terminal and checking whether it is accepted or not is determined by recursive function.

B.Tree

In mathematics, and more specifically in graph theory, a tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any acyclic connected graph is a tree. A forest is a disjoint union of trees.Terms commonly seen regarding tree are mentioned below.

- Leaf
  A vertex of a tree is called a leaf if it has no children.
- Rooted Tree
  A rooted tree is a tree in which one vertex has been designated as the root and every edge is directed away from the root
- Parent
  Suppose that T is a rooted tree. If v is a vertex in T other than the root, the parent of v is the unique vertex u such that there is a directed edge from u to v.
- Child
  If U is the parent of v, then v is called a child of u.
- Siblings
  Vertices with same parent is called siblings.
- Subtree
  If a is a vertex in a tree, the subtree with a as its root is the subgraph of the tree consisting of a and its descendants and all edges incident to these descendants.
- Internal Vertices
  Vertices that have children is internal vertices.
- M-ary Tree
  A rooted tree is called an m-ary tree if every internal vertex has no more than m children. The tree is called a full m-ary tree if every internal vertex has exactly m children.
- Binary Tree
  A m-ary tree with m = 2 is called binary tree.This kind of tree is commonly used in computer science.
- Level
  The level of vertex v in a rooted tree is the length of the unique path from root to this vertex
- Height
  The maximum level of vertices is the height of the tree.
- Balanced
  A rooted m-ary tree is balanced if all leaves are at level height or height-1.
- Ordered Root Tree
  An ordered root tree is a rooted tree where the children of each internal vertex is in order.

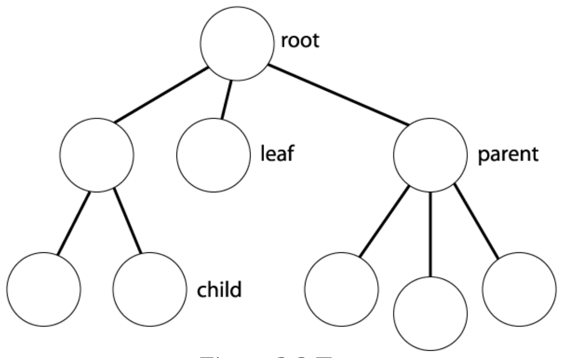


Figure 2.2 Tree
(Source : https://i.stack.imgur.com/5kJXf.gif)

Tree is closely related to recursive function,especially binary tree.Binary tree is a recursive structure due to each node have branch(s) that is also a tree. Every branch of tree is called subtree[2].

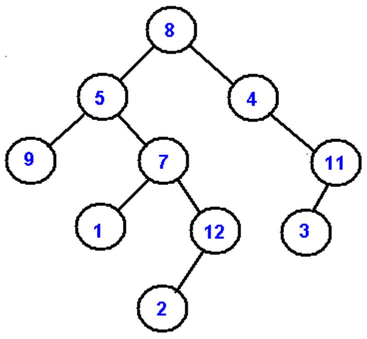


Figure 2.3 Binary Tree
(Source : https://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/pix/tree1.bmp)

As we introduce earlier,recursive function consists of basis and the recursive step.In binary tree's problem the basis should be empty tree is a binary tree.The recursive step should be the subtree of a binary tree in which is also a binary tree too.

This binary tree have play an important role in building a compiler.A terminal is node in a binary tree.Later,we will see that our goal is to manipulate this tree to see if a certain program have an error or not.

C.Grammar

In the literary sense of the term, grammars denote syntactical rules for conversation in natural languages.Grammar G can be formally written as a 4-tuple (N, T, S, P) where :

- **N** or $\mathbf{V}_N$ is a set of variables or non-terminal symbols.
- **T** or ∑ is a set of Terminal symbols.
- **S** is a special variable called the Start symbol, S ∈ N
- **P** is Production rules for Terminals and Non-terminals. A production rule has the form α → β, where α and β are

strings on $V_N \cup \sum$ and least one symbol of α belongs to V_N.

Strings may be derived from other strings using the productions in a grammar. If a grammar **G** has a production **α → β**, we can say that **x α y** derives **x β y** in **G**. This derivation is written as $\boldsymbol{x\,\alpha\,y \Rightarrow x\,\beta\,y}$**.**

A context-free grammar (CFG) is a set of recursive rewriting rules (or *productions*) used to generate patterns of strings.[3] A CFG consists of the following components:

- a set of *terminal symbols*, which are the characters of the alphabet that appear in the strings generated by the grammar.
- a set of *nonterminal symbols* or *variable*, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.Normally used surrounded by ‘< >’.For instance <real> define variable real.
- a set of *productions*, which are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production).
- a *start symbol*, which is a special nonterminal symbol that appears in the initial string generated by the grammar.

A CFG’s production describing real numbers in Pascal is show below.Noted that ‘|’ means ‘or’ and ε means empty string.

1. <real> → <digit> <digit*> <decimal part> <exp>
2. <digit*> → <digit> <digit*> | ε
3. <decimal part> → '.' <digit> <digit*> | ε
4. <exp> → 'E' <sign> <digit> <digit*> | ε
5. <sign> → + | - | ε
6. <digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A CFG is in Chomsky Normal Form(CNF) if the productions are in the following forms: A → a ; A → BC ; S → ε where A,B,C are nonterminal,a is a terminal,and S is a start symbol.Algorithm to convert any CFG to CNF is :

1. If the start symbol S occurs on some right side, create a new start symbol S’ and a new production S’→ S.
2. Remove Null productions (production that derive ε)
3. Remove unit productions (production that derive just one nonterminal)
4. Replace each production $A \rightarrow B_1 \ldots B_n$ where n > 2 with $A \rightarrow B_1C$ where $C \rightarrow B_2 \ldots B_n$. Repeat this step for all productions having two or more symbols in the right side.
5. If the right side of any production is in the form A → aB where a is a terminal and A, B are non-terminal, then the production is replaced by A → XB and X → a. Repeat this step for every production which is in the form A → aB.

Deriving CFG and CNF is not alike.Some method require us to convert any non-CNF grammar to CNF,so the derivation could work.For instance For instance, CYK Algorithm(Cocke-Younger-Kasami).Other method does not have boundaries whether to have a CFG or CNF as it can derives all.The example for this is the recursive descent parser.

In grammar , there are two ways to derivate production.That is Leftmost derivation and Rightmost derivation[5]. Now consider the grammar G = ({S, A, B, C}, {a, b, c}, S, P) where P = {S → ABC, A→ aA, A→ ε, B→ bB, B→ ε, C→ cC, C→ ε }.

The leftmost derivation will derive the terminal as show below.

1. S → ABC
2. ABC → aABC
3. aABC → aABcC
4. aABcC → aBcC
5. aBcC → abBcC
6. abBcC → abBc
7. abBc → abbBc
8. abbBc → abbc

Otherwise, the rightmost derivation will derive the terminal as show below.

1. S → ABC
2. ABC → ABcC
3. ABcC → ABc
4. ABc → AbBc
5. AbBc → AbbBc
6. AbbBc → Abbc
7. Abbc → aAbbc
8. aAbbc → abbc

Different derivations result in quite different sentential forms, but for a CFG, it really does not make much difference in what order we expand the variables.But commonly in practice,we use the leftmost derivation technique.Recursive Descent Parser use leftmost derivation technique.

III. RECURSIVE DESCENT PARSER

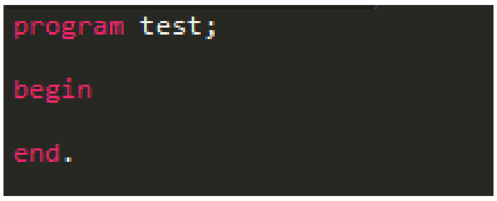
A parser is a program that determines the grammatical structure of a phrase in the language/grammar. This is the first step in determining the meaning of the phrase,which for a programming language means translating it into machine language.[4]

The parsing method that we will mention is recursive descent parsing with a back-tracking.Noted that not any grammar can be derived from recursive descent parsing.They have to satify certain properties.Recursive descent parsing is a top-down parsing that build tree from root symbol.

Each production corresponds to one recursive procedure.Each procedure recognizes an instance of a non-terminal, returns tree fragment for the non-terminal.The method of the this method is recursively check for the grammar and derive the syntax from Start to all terminals.

If one rule in grammar can not derive all terminals,it will go back-tracking to find other rules that can satisfies.If all rules have been implemented yet there still no rules that can do the derivation,our compiler deduce that the program is not accepted.

In order to do parsing,we need several things to do.First we need to determine certain grammar that satisfies one programming language.Our RDP will parse according to this grammar.This grammar does not have to be CFG or CNF as it can parse through both of them.The next thing we need is a tokenizer.Event though tokenizer seems optional,tokenizer translate terminals to one alphabet for speeding up purposes.



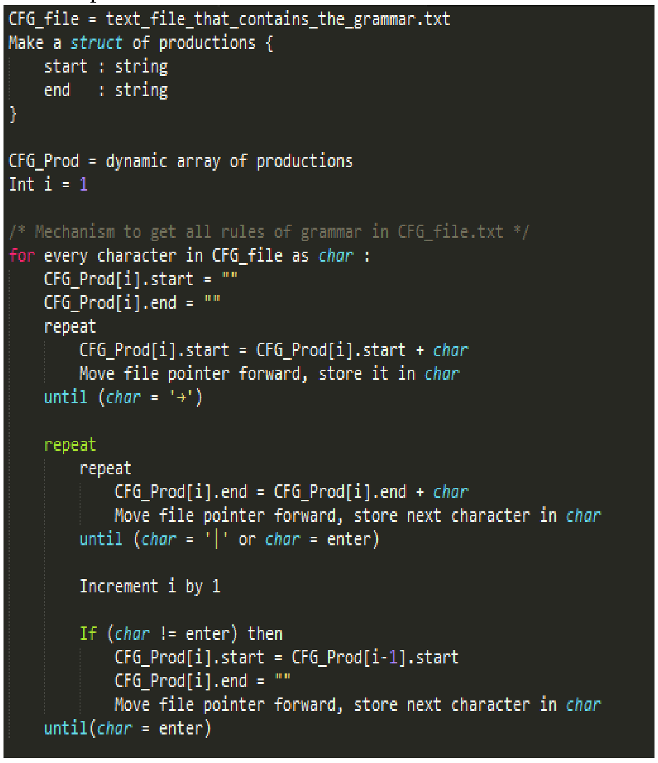
```
program test;

begin

end.
```

Figure 3.1 Simple pascal program that we want to parse (Images taken by author)

Let us take an example based on Figure 3.1.First of all,we need to make grammar regarding that example.

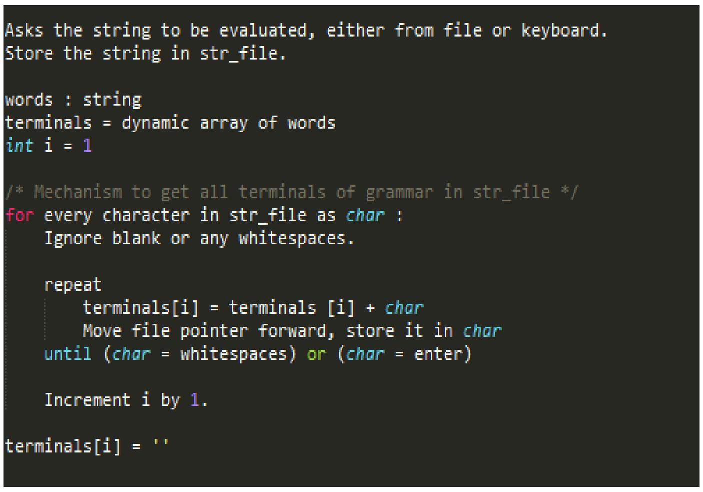
1. <start> → <header> <var> <main_body>
2. <header> → program <identifier> ;
3. <var> → var <identifier> <identifier> <more_var> <colon> <var_type> <semicolon> | ε
4. <more_var> → <comma> <identifier> <more_var> | ε
5. <main_body> → begin <code> end.
6. <code> → ε
7. <identifier> → <0..9> <identifier> | <a..z> <identifier> <A..Z> <identifier> | ε

We restrict the <code> production to an empty for now.If we want to develop program mechanism such as assignment variable,for-loop,while-loop,if-else , we could make a grammar regarding each mechanism.The pseudo code for the recursive descent parses is shown below.



```
CFG_file = text_file_that_contains_the_grammar.txt
Make a struct of productions {
    start : string
    end   : string
}

CFG_Prod = dynamic array of productions
Int i = 1

/* Mechanism to get all rules of grammar in CFG_file.txt */
for every character in CFG_file as char :
    CFG_Prod[i].start = ""
    CFG_Prod[i].end = ""
    repeat
        CFG_Prod[i].start = CFG_Prod[i].start + char
        Move file pointer forward, store it in char
    until (char = '→')

    repeat
        repeat
            CFG_Prod[i].end = CFG_Prod[i].end + char
            Move file pointer forward, store next character in char
        until (char = '|' or char = enter)

        Increment i by 1

        If (char != enter) then
            CFG_Prod[i].start = CFG_Prod[i-1].start
            CFG_Prod[i].end = ""
            Move file pointer forward, store next character in char
    until(char = enter)
```

Figure 3.2 Pseudo code for getting grammar production stored in CFG_file (Images taken by author)



```
Asks the string to be evaluated, either from file or keyboard.
Store the string in str_file.

words : string
terminals = dynamic array of words
int i = 1

/* Mechanism to get all terminals of grammar in str_file */
for every character in str_file as char :
    Ignore blank or any whitespaces.

    repeat
        terminals[i] = terminals [i] + char
        Move file pointer forward, store it in char
    until (char = whitespaces) or (char = enter)

    Increment i by 1.

terminals[i] = ''
```

Figure 3.3 Pseudo code for getting all terminals in str_file (Images taken by author)

To understand this recursive descent parser,let us visualize it by drawing the parse tree.Understand the terminal that we want to derive is program test ; begin end.

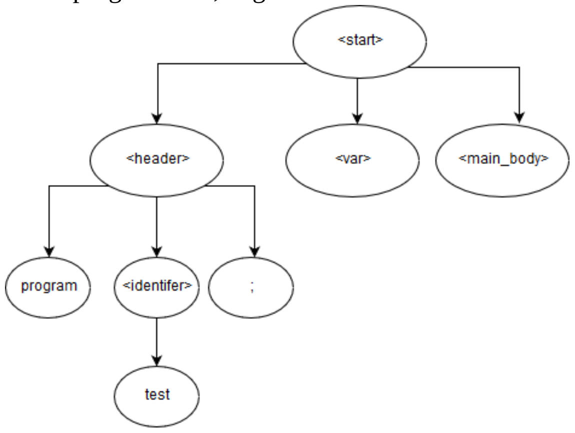


Figure 3.4 Checking the derivability of the production (Images taken by author)

From the start variable it will try the first derivation,which is <header><var><main_body>.The header part will derive program immediately.The first terminal can be generated from header so it will advance to other terminal without backtracking.<Identifier> could derived test and also will advance to semicolon and also accepted.

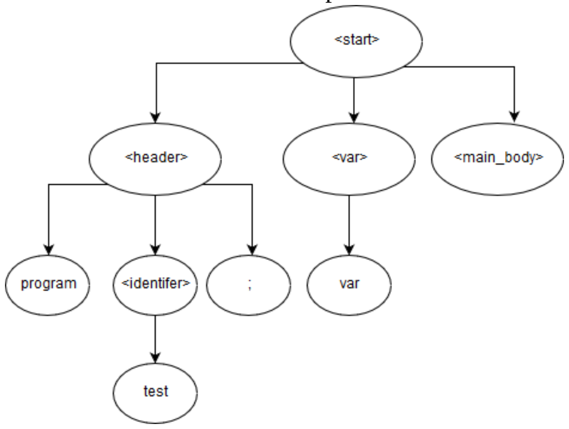


Figure 3.5 Example of back-tracking
(Images taken by author)

As it goes to <var> production ,first production will derive var and it's not what current terminal is.So it will backtrack and search for another <var> production.

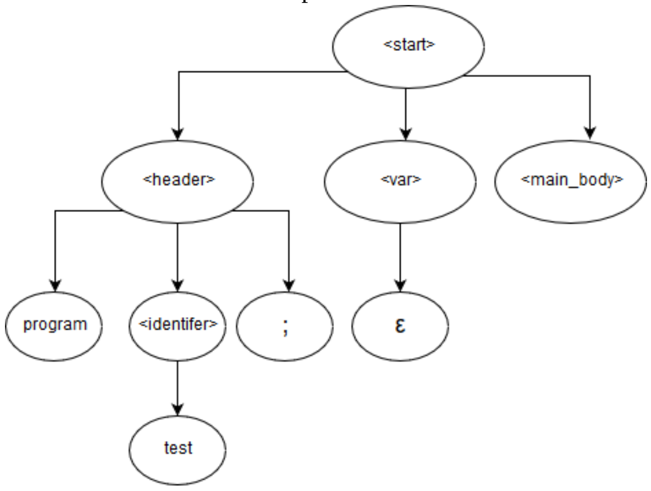


Figure 3.6 Skipping empty string production / ε
(Images taken by author)

When the production meet empty production,it will skip to the next production with still the same current terminal.

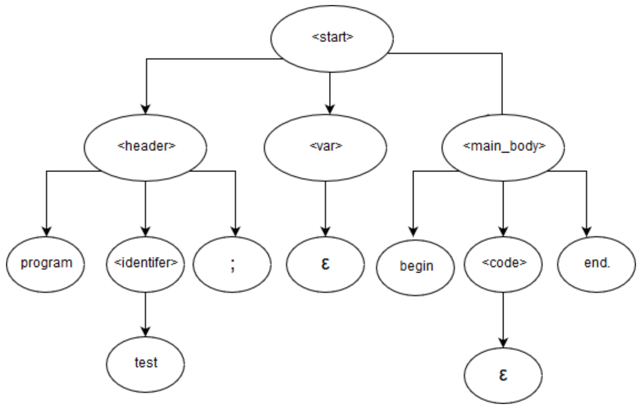


Figure 3.7 Finishing the parsing
(Images taken by author)

It will go to <main_body> derivation and find 'begin' and it matched with the current terminal.It will advance again to <code> and find empty production and skip it again.It will advance to meet 'end.' .As soon as it has parse the whole terminal,our program will return true and conclude that the syntax is accepted.

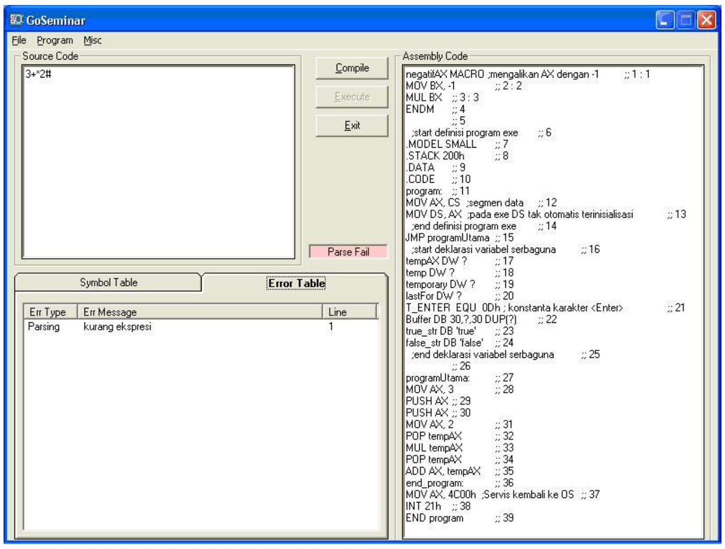
Recursive is used to searching the non-terminal derivation.If the derivation is still consist of variable(non-terminal),it will call the function again until it meet a terminal.

The other key in this parsing is the idea of backtracking.If the derivation do not match the current terminal,it will search for other possibilities and if it can not be backtracked anymore,that is when a certain program concluded not accepted.

IV. APPLICATION

Recursive Descent Parser is used in many application.In order to make a more complex compiler, the method of this parsing need to be more efficient and also the CFG need to be more complex as the rules increases.The parsing strategy is used not only to make a compiler,but also for another string parsing.

For instance,a calculator check expression could be derive from the Recursive Descent Parser.The grammar should be changed due to different purpose . After determining the grammar,we need a function that could calculate the operation based on token (+,-,*,/,div,mod).



```
3+*2#
```

Err Type	Err Message	Line
Parsing	kurang ekspresi	1

```
negatifAX MACRO ;mengalikan AX dengan -1        ;; 1 : 1
MOV BX, -1          ;; 2 : 2
MUL BX     ;; 3 : 3
ENDM       ;; 4
           ;; 5
  ;start definisi program exe       ;; 6
.MODEL SMALL       ;; 7
.STACK 200h        ;; 8
.DATA      ;; 9
.CODE      ;; 10
program:   ;; 11
MOV AX, CS  ;segmen data     ;; 12
MOV DS, AX  ;pada exe DS tak otomatis terinisialisasi          ;; 13
  ;end definisi program exe       ;; 14
JMP programUtama  ;; 15
  ;start deklarasi variabel serbaguna        ;; 16
tempAX DW ?         ;; 17
temp DW ?           ;; 18
temporary DW ?      ;; 19
lastFor DW ?        ;; 20
T_ENTER  EQU  0Dh ; konstanta karakter <Enter>           ;; 21
Buffer DB 30,?,30 DUP(?)       ;; 22
true_str DB 'true'      ;; 23
false_str DB 'false'    ;; 24
  ;end deklarasi variabel serbaguna        ;; 25
           ;; 26
programUtama:       ;; 27
MOV AX, 3           ;; 28
PUSH AX ;; 29
PUSH AX ;; 30
MOV AX, 2           ;; 31
POP tempAX          ;; 32
MUL tempAX          ;; 33
POP tempAX          ;; 34
ADD AX, tempAX      ;; 35
end_program:        ;; 36
MOV AX, 4C00h  ;Servis kembali ke OS   ;; 37
INT 21h   ;; 38
END program         ;; 39
```

Figure 4.1 Application of RDP in calculator check expression (Source : http://basitadhi.blogspot.co.id/2009/10/pembuatan-compiler-dengan-metode.html)

V. CONCLUSION

Building a simple compiler could be done with the help of recursive concept in recursive descent parser.Not only in generating a compiler,the recursive descent parser could be used in many other application too.The algorithm for recursive descent parser is discussed and implemented in this paper.However,this implementation leaves much room for improvement as it is still lack of efficiency.

VI. ACKNOWLEDGMENT

The first and foremost thanks from the author is to God for providing the author inspiration,time,and facility to be able to write and finish this paper.Special acknowledgment to Dra Harlili S.M.Sc as the lecturer of the author's Mathematical Discrete class,for guidance in preparing this paper.Last but not least,the author took credits for author's parents who always supporting the author's education in Bandung Institute Technology.

REFERENCES

[1] Zery, *Recursive Descent Parser*. Available: http://duniazery.blogspot.co.id/2013/04/recursive-descent-parser-rekursif.html. (Retrieved November 26, 2017 ,21:16)
[2]Rinaldi Munir,*Diktat Kuliah IF2110 Matematika Diskrit*, Informatics Undergraduate Program School of Electrical Engineering and Informatics
Bandung Institute of Technology,2006.
[3]Nelson,*Context Free Grammar*.Available: https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html (Retrieved December 1, 2017 ,05.12)
[4]http://math.hws.edu/javanotes/c9/s5.html (Retrieved December 1, 2017 ,16:57)
[5] Matuszek ,*Leftmost and Rightmost Derivation* .Available in: https://www.seas.upenn.edu/~cit596/notes/dave/cfg8.html (Retrieved December 2, 2017 ,19:43)

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2017

[illegible]

Manasye Shousen Bukit - 13516122