1. Introduction

DNF Converter is a program that transforms a propositional formula into a Disjunctive Normal Form (DNF). DNF is a two-level propositional formulas which has form of $(a11 \wedge a12 \wedge \cdots \wedge a1n1) \vee (a21 \wedge a22 \wedge \cdots \wedge a1n2) \vee \ldots (am1 \wedge am2 \wedge \ldots \wedge amnm)$ where aij is p or $\neg p$ for an atomic propositional variable p.
Every propositional formula has an equivalent DNF form. So our program will consume the propositional formula and produce the result through several procedures. 2. Approach When given a propositional formula, program should change and print it in DNF format. All input must be prefixed in the parentheses. At output expression, the numbers in the same row are all connected by conjunction, and each row is connected by disjunction. And the program must print out an error message if the given input does not follow that rule.

2.1) Solution Design

Our DNF Converter follow this procedure to solve the problem. Each procedure has detailed description to help reader understand.

[1] Parse the proposition formula First, input will be split by space so that each expression goes into binary tree structure. For example, if the propositional formula (and (not a1) (*or* a2 *a*3)) converts into tree structure.
Each propositional variable will be a node and has it's own value. (AND : 1, OR : -1, NOT 0) When each expression goes into binary tree, there are some rules. First, if the node data is zero, the subtree must be created on the left node. Second, if a node data has a value 1 or -1, it has to check if the left node or right node is null. If the left node has a null value, make a subtree on the left node, and if the left node is full, go to the right node and make a subtree.

[2] Negate all child nodes of 'NOT' nodes and delete it After the all variable goes into tree, the NOT node must be eliminated. Before that, all child nodes of NOT node must be negated. AND becomes OR, OR becomes AND, and each numerical variable will be negative integer. After all nodes negated, NOT node will be deleted in tree.

[3] Apply the distributive law

Applying distributive law, we should make the form of conjunctive clauses. So, to do that, we must place OR node to the root, and apply the distributive law. The left child of root node will be bound with child nodes of OR node.

[4] Interpret the final expressions and print the result. Now we can extract conjunctive clauses from tree. All numerical value node who has AND node for the parent will be the line of output result. With the graph above, the output will be a1, for first line and a2, -a4 for second line, a3. -4 for the last line.

[5] Print all Truth Values
After all expressions interpreted, all number node must be stored in Array and printed. It will print a1, a2, a3 and -a4.

3. Results

For the right input,

```
(or a1 (not (or (not (or a2 a3)) a4)))
```

It will return the list of conjunctive clauses at the end of program.

```
1
2 -4
3 -4
0
1 2 -4 3
```

But, for the wrong input like incorrect parenthesis input, It will return error message for the wrong input that isn't propositional formula.

4. Discussions

The lesson we learned from this problem was it is really hard to make parser and interpreter of new expressions. The program we made is not perfect. Some input works perfectly, but some doesn't. We tried to make algorithm that converts certain expression to the value from several example expressions. But in that way, it couldn't cover all expressions. We needed some grammar that cover all kinds of expressions. From that point, it was like making new language interpreter. Propositional logic must be converted into binary tree and by its own grammar and finally it was interpreted. It was little bit different with coding that we did before. Because we had to handle new expressions, it was not easy to make perfect grammar that cover all expressions. It was really good chance to experience this kind of problem.