Lab 1 - Truth Tables and Logic Expressions

1. **Introduction**: My lab has two parts: The base program which takes a truth table as an input and outputs a sum of products. And a second independent program that takes a logical expression and outputs a truth table. The second part is for extra credit and represents most of the code and complexity of the assignment.
2. **Process**: For the base assignment I used a nested for loop to swap zeros and ones with letters and nots. This was achieved with a nested for loop. The outer for loop separates the user input along commas into chunks. The inner for loop compares each character against the F value and prints a backslash if the characters do not match and prints an incrementing letter for each character except the last.

    for (i, chunk) in entire\_input.trim().split(',').enumerate() { // for each comma separated chunk

        if i > 0 { // not first run

            print!(" + ");

        }

        for (j, c) in chunk.chars().enumerate() { // compare each character to the last char in the string

            if j < chunk.len() -1 { // ignore last char

                if chunk.ends\_with(c) == false {

                    print!("/");

                }

                // Print j'th letter in alphabet

                print!("{}", (j as u8 + 'A' as u8) as char);

            }

        }

    }

**The explanation for my extra credit implementation is at the end of the document.**

1. **Testing**: To test my base program, I ran a variety of inputs, see below. Invalid characters are ignored, and the program throws an error if any substring has more than 5 Booleans to enforce the not going beyond D limit. My extra credit is more sensitive, there are many places where invalid expression formatting can cause errors and while crashes are caught the program simply continues without printing a table. However, some tricky cases are handled well and provide the expected behavior, such as multiple parenthesis around a single value, multiple nots in a row, and not before a parenthesis.

**Testing examples**:

No crashes, invalid characters are simply ignored.



Crashes on purpose, as a variable beyond D was given



A black background with white text

Description automatically generated

1. **Results**: A string from the user is separated along commas, the variables needed (ABC…) are determined by the substring length. Each value besides, the last value F, is printed and if it is not equal to F it is not-ed as well. The result of each substring is printed with a plus between them.



For the extra credit, a logic expression is entered. It must strictly follow the assignment specifications and cannot include an invalid characters or whitespaces. The output is a truth table generated by substituting all combinations of the Boolean variables and solving.

The expression ((A+B+C\*/(D+/D)+A)+(B\*/C)) program can simplified to: if A or B and not C, which we see the correct results for.

A screen shot of a computer code

Description automatically generated

1. **Conclusion**: I started with the extra credit, so some of its code isn’t very well written because I was still learning rust. But the main program is pretty well written all the assignment runs well. In my testing I always get the expected results. In future assignments I will use error handling better. I used panic like throw in C++, but panic is only meant to be used for unrecoverable errors and will always print a full stack trace even if caught.
2. **References and Acknowledgements**: I used stack overflow, the official rust documentation, and ChatGPT to help me learn rust and complete the assignment. All code is my own, I borrowed formatting and approaches from many sources but did not copy and paste code from any source. To give an example of how I used ChatGPT, I wanted to filter character out of a string but I wanted to give characters to keep not remove, so I asked ChatGPT. ChatGPT gave me:

user\_input.retain(|c| c.is\_numeric() || c == ',');

But after looking up *retain()* in the official rust documentation, my final code looks like this:

user\_input.retain(|c| c == '1' ||  c == '0' || c == ',');

<https://chat.openai.com/share/06766d1e-f4d0-4162-944d-aa7d87f32d38>

**Extra credit process**:

I’m giving a high-level overview of the extra credit as it is complicated enough to take multiple pages to explain and my implementation is not very elegant, so going into implementation detail will only add confusion. Below are most of the functions followed by their call order. A few utility functions have been omitted for simplicity.

fn run\_extra\_credit(expression: String)

fn substitute\_variables(expression: String, variables: &[bool], var\_count: usize) -> String

fn apply\_nots(not\_char: char, expression: String) -> String

fn perform\_operation(expression: String) -> String

fn op\_max\_depth(expression: &str) -> i32

fn remove\_single\_parenthesis(expression: String) -> String

1. **main** calls **run\_extra\_credit**
2. **run\_extra\_credit** does the following **for every combination in the truth table**:
   1. **substitute\_variables** This replaces A,B,C… with zeros and ones, every combination of zeros and ones will be called.
   2. **apply\_nots**. Variables are inverted if they are preceded by an odd number of nots, nots before parenthesis are left in place
   3. **While the expression is not solved**: This loop calculates one operation and exits once a single zero or one remains.
      1. **Remove parenthesis**. This removes the parenthesis around single variables ex: “(1)” becomes “1”. This function is recursive and will remove multiple parenthesis, ex: “(((0)))” becomes “0”.
         1. **apply\_nots** is applied. so ‘/(0)’ becomes ‘1’
      2. **perform\_operation** returns the input expression but with the inner most operation solved. Ex: “(1 and **(0 or 1)**)” => “(1 and **(1)**)”.

Now what does this look like with an actual expression? Using English instead of the programs notation, let’s examine (not A and not (B or C))

(not A and not (B or C))

Our first step comes at 2.a where we substitute numbers for variables. Every combination of variables will be run in the program but let’s look at A=0, B=0, C=1. Our expression becomes:

(not 0 and not (0 or 1))

Moving on to 2.b we apply nots, remember nots proceeding parenthesis are left in place. Note the red 1 has been not-ed but the green not hasn’t changed.

(1 and not (0 or 1))

2.c.i has no effect as there are no parenthesis to remove or nots to apply.

2.c.ii calculates the inner most operation. “0 or 1” => “1”

(not 0 and not (1))

2.c.i removes the parenthesis and nots the second variable.

(1 and 0)

2.c.ii calculates. “0 and 1” => “0”

(0)

(0) is considered solved so the loop breaks. The remaining parentheses are removed, and the value is printed to the truth table in a column next to the variable values. A=0, B=0, C=1 with a result of 0 gives us: **0 0 1 | 0**