Programming Languages

Final Project

Francisco A. Díaz Vergara A01204695
Alberto Matute Beltrán A01704584

Professor: Benjamín Valdés Aguirre

Implementing truth table generator in C

Contents

Introduction

For this project, we decided to implement the **logic paradigm**. Our goal was to show how our implementation demonstrates how the logic paradigm expresses facts and rules about problems with a system of formal logic. We utilized C as our programming language since C is a multi-paradigm language and offers a vast level of creativity and flexibility on approaching a problem and how to solve it. We utilized visual aids to show the final result.

Context of the problem

Logic programming is a programming paradigm which is largely based on formal logic. Any program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain.

A truth table is a mathematical table used in logic specifically in connection with Boolean algebra, boolean functions, and propositional calculus which sets out the functional values of logical expressions on each of their functional arguments, that is, for each combination of values taken by their logical variables.[1] In particular, truth tables can be used to show whether a propositional expression is true for all legitimate input values, that is, logically valid.

Algorithm

For the algorithm first, we defined what operators can be used when evaluating the expression given by the user. We created a set of rules that are always true, for example; We defined **~** as the **NOT** operator, which makes expression **A** false (!A). We defined the operator in the following way:

~ as the operator NOT (!A),

& as the operator AND (A && B),

| as the operator OR (A || B),

> as the implication operator (!A || B),

= as the equal to operand (A == B).

Example:

given the expression T & (T | F)

1. search the expression, save the operators in a “operator stack” and the variables in a “variable stack”.
    a. Operator Stack

&	(	|	)

    b. Variable stack

T	T	F

2. analize hierarchy and evaluate based on position
    a. Operator Stack

    Because of parenthesis, evaluate operation inside first

    The operator “|” translated by the tool as “OR” requires 2 variables.

&	(	|	)

    b. Variable stack

    Variables required for “OR” operation

T	T	F

3. Transform operation for tool
    a. RES = T | F = T;
4. Insert value into Variable stack
    a. Variable stack

    Variable inserted

T	T

5. Solve next operation in stack
    a. Operator Stack

    The operator “&” translated by the tool as “AND” requires 2 variables.

&

    b. Variable stack

    Variables available for “AND” operation

T	T

6. Transform operation for tool to solve
    a. RES = T & T = T
7. Insert value in Variable stack
    a. Variable stack

    Variable inserted

T

8. Because there are no more operators in the Operator stack, return last value of Variable stack in table such as
    a.

T & (T | F)	T

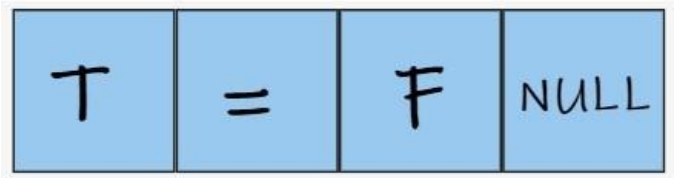
Implementation

For the implementation of the algorithm, we utilized "simulated stacks." In other words, we used arrays to store every value and operator that the user inputs. We created an array for operators and an array for values. Once we had these arrays, we iterated through them from start to finish. For example:

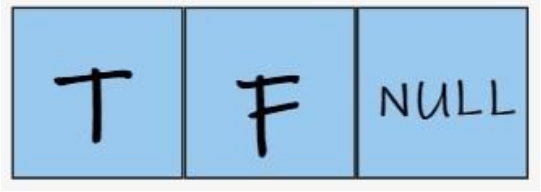
Let's say we have a simple expression such as **T = F**. We receive this expression as an array and check if the value we're currently at is an operand or letter from the alphabet. We iterate through the array while the current value is different from NULL.
image 1

First, our expression array would be in position 0 **exprArr[0],** containing **T**. Then, we store T in **valArr[0]** and move to position 1 of the values array. If we continue iterating through the expression array, we'll find a space. We designed the program so that if the current value of the expression array is a space, the program can move on to the next value since a blank space does not help determine the outcome of the expression.
Next, we'll find operand **=,** storing it in position 0 of the operands array **opArr[0]**, then we find another space, so we move on to the next position. Once we get to the final position, which contains value **F,** we store that value in the current position of the array, which would be **valArr[1].** Now we're ready to implement our algorithm utilizing arrays **opArr[]** and **valArr[].**

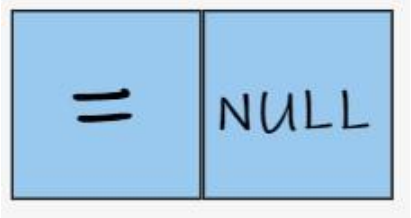
Expression Array (simulated stack)



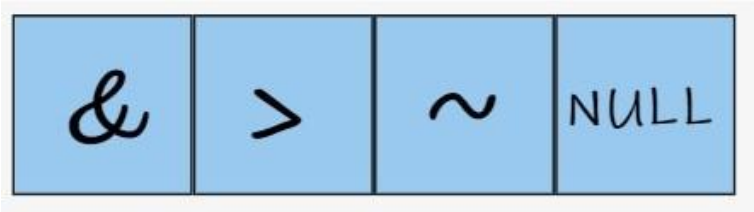
valArr[]:



opArr[]:



Another example of a more complex stack of operators:



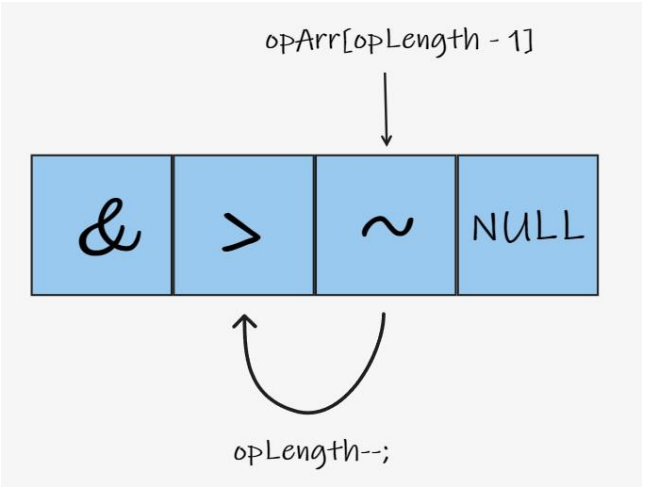
Once we have our arrays filled with data, we can iterate through them from end to beginning to simulate a stack. We begin the iteration at the length of each array -1 (length - 1) to be at the last value or operator we found. We utilize a flag to determine if the last value of the array (the one we're currently checking) is a valid character, such as a letter from the alphabet or one of the valid operators we defined previously. If the value is valid, we set the flag to true and values **a** and **b** to 0. These variables will then be assigned to the values found in the **valArr[].**

Now we check the last value in the operators array. If the value is **~ (NOT),** then we only set one of the variables (a or b) to the last value from the **valArr[]** because the other value won't determine the final result since it will be affected by the **~** operator. We do valArr[position - 1] so that we don't consider the other value in the expression.

Code representation:

```
if(opArr[opLength - 1] == '~')
{
    if(valueLength < 1) //there has to be 2 values
    {
      return 0;
    }

    a = valArr[valueLength - 1];
    valueLength--; //delete that value since it's already assigned
}
```



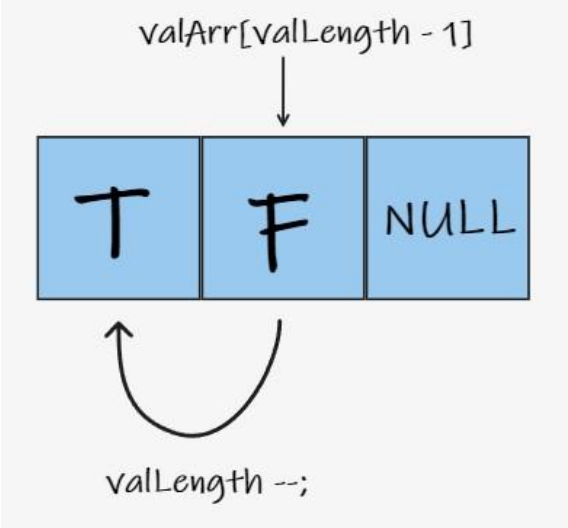
However, if the operator found is not the ~ operator, we need to consider both values (a and b) to calculate the result correctly.

```
else
{
   if(valueLength < 1) //if there's no values
   {
     return 0;
   }
   b = valArr[valueLength - 1];

   valueLength--; //delete that value from the stack

   a = valArr[valueLength - 1];

   valueLength--; //delete that value from the stack
}
```



Once we have successfully validated the expression inputted by the user then we can move on to applying our logic algorithm and finally getting our final result.

First we created a “valid operators array” which we’ll use to compare with the operators given by the user. Once everything checks out we call our calculator function and apply the following logic.

```
switch(operador)
   {
      case '~': return (!a);
      case '&': return a & b;
      case '|': return a | b;
      case '>': return (!a) | b;
      case '=': return ((!a) | b) & ((!b) | a);
   }
```

We utilized a switch statement to utilize the different operators depending on the one we're currently using. We utilize this logic several times combined with the "simulated stacks" logic to go through all the different expressions, including parenthesis, operators and values. Lastly we utilize for statements to iterate through all the data we got to print the values, operators and their corresponding results so that the user can examine all the different results in a visual way.

Installation

Step 1 :

- Open terminal (Ubuntu) sudo apt-get update
- sudo apt-get upgrade
- sudo apt-get install git
- sudo apt-get install build-essential
- git clone https://github.com/zoelabbb/conio.h.git
- cd conio.h

Step 2 :

- After you finish download file conio.h
- Copy **file** conio.h
- Go to /usr/include/
- Right click on folder /usr/include/
- Choose Open as Administrator
- Paste file conio.h
- Close your IDE and open again

Step 3:

- $ gcc main.c -o main
- $ ./main
- follow instructions in the menu of the tool for a better understanding of the tool.

Conclusion

Truth tables can be used to prove many other logical equivalences, logic programming can be viewed as controlled deduction. An important concept in logic programming is the separation of programs into their logic component and their control component. With pure logic programming languages, the logic component alone determines the solutions produced. The control component can be varied to provide alternative ways of executing a logic program.

When working on this project we realized (the hard way), that designing algorithms is oftentimes more complicated than implementing your algorithm in code, this course gave us a closer look at all the different programming paradigms and how different programming languages serve their purpose for certain algorithms better than others. Each programming paradigm has their own way of working and their own way of processing data and experiencing different paradigms was a great way to increase our understanding of programming as up and coming software engineers.

Bibliography

GeeksforGeeks. (06 Sep, 2021). Expression Evaluation. 11/1/2021, de GeeksforGeeks Sitio web: https://www.geeksforgeeks.org/expression-evaluation/