Problem Statement

The project works upon the variable conditions and evaluates them to determine the success cases required to achieve the final output. All possible permutations of the logical expressions are considered and a truth table is designed.

It then moves on with assessing infix expression, converting it to postfix and displays a waveform of this output expression in accordance with the truth table in the terminal.

Analysis of Problem

In this project, we aim to implement logical expressions we have learnt, using the concepts of Data structures and algorithms. This project integrates both concepts and creates real world applications of a program.

In our code, we use the inherent concepts of stacks as linked lists and use this technique for expression evaluation and simulation.

The stacks are dynamically implemented i.e. using linked lists. This implies that memory is dynamically created when the code requires it to be allocated, and there is no wastage of memory space by our program.

The code takes an infix expression, converts it to postfix and displays it. It asks the user what every variable means and then displays a truth table of every possibility the expression can have, running every possible case during simulation.

It then displays a waveform of this output expression in accordance with the truth table in the terminal and using the concepts of file handling in C, saves this output in a file for future use.

Data structure and Algorithmic technique

A. Linked-List Implementation of Stacks

B. Infix to Postfix Conversion

Linked-List Implementation of Stacks

One disadvantage of using an array to implement a stack is the wasted space - most of the time most of the array is unused. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A more elegant and economical implementation of a stack uses a linked list, which is a data structure that links together individual data objects as if they were links in a chain of data.

Stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as 'top' element. That means every newly inserted element is pointed by 'top'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by 'top' by moving 'top' to its next node in the list. The next field of the first element must be always NULL.

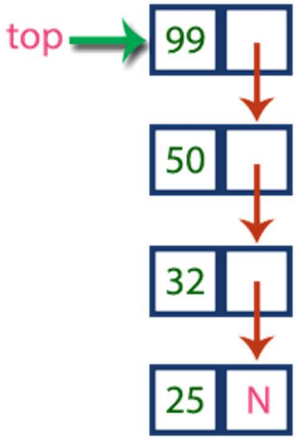

Fig 1: Stack representation

In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32, 50 and 99.

Infix to Postfix Conversion

Infix expression: The expression of the form a op b. When an operator is in-between every pair of operands.

Postfix expression: The expression of the form a b op. When an operator is followed for every pair of operands.

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left.

Consider the below expression: a op1 b op2 c op3 d

If op1 = +, op2 = *, op3 = +

The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then added to d after another scan.

The corresponding expression in postfix form is: abc*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

Algorithm

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, output it.

3. Else

3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty), push it.

3.2 Else, Pop the operator from the stack until the precedence of the scanned operator is less - equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.

4. If the scanned character is an '(', push it to the stack.

5. If the scanned character is an ')', pop and output from the stack until an '(' is encountered.

6. Repeat steps 2-6 until infix expression is scanned.

7. Pop and output from the stack until it is not empty.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Fig 2: Example of infix to postfix

Algorithm

START

1. Include all header files and standard library functions.
2. Define MAX value as 100.
3. Initialize all required variables for the code to be used with appropriate data types.
4. Create a structure node, which acts as a node for the linked list in dynamic implementation of stacks and stores data and has pointer next.
5. Initialize a pointer l1 to NULL.
6. Create function insert to push the element into the stack (linked list) and change the value of the top pointer respectively.
7. Create function delet, to pop the element from the stack in the linked list.
8. Create function display, to display all the elements of the linked list with the help of its pointer and display all the elements present in the stack.
9. Create a function prior, which takes in the symbol input and evaluates it and assigns it a priority to be ordered and removed or not from the stack.
10. Create a function cnvrt, to take the input string of the expression and convert it to a form which can be used for evaluation of the given string i.e. postfix expression. This conversion also takes place using the concept of dynamic implementation of stacks using linked list.
11. Create a function count0, to check the uniqueness of a variable in the expression to be evaluated to give a more minimalized output.
12. Create a function no_of_variables, to be able to count the number of unique variable in the expression checked already from the previous function in step 12.
13. Create a function ttable, to initialize a 2 dimensional array to be able to save the input conditions and the output cases in the array to form the truth table.
14. Create a function ttabledisplay, which displays the truth table created in the function in step 14 and displays it with appropriate labeling and bordering and aesthetics.
15. Create a function eval, which takes the previously converted input expression to postfix expression and now evaluates it using the concept of stacks in dynamic memory implementation using linked lists.
16. Create a function graphdisplay, which take the values in the truth table and depending on the frequency input by the users. Now depending on the value 1 or 0 the graph show – or _

respectively a gives and equivalent waveform response for the given expression and hence proves the validity of the given expression.

17. Create a function graphdisplay_out, which uses the concepts of file handling in C to write the output and waveform into a text file to save the result and use it for future implementations.
18. Next create a fn. success_cases, which takes the data of the truth table and checks all the outputs where the output is a success case i.e. 1. It then display all the parameters of such cases i.e. variable input cases.
19. Also another function meaning_of_variable is also created which asks the user what every variable in their input expression means and saaves this meaning of the variables.
20. Later another function final_saying, uses these meanings input by the user to display what the final answer of the input expression would be based on the meaning and the truth cases of evaluation.
21. A function licence_check() is also created to properiate the software program. Before the beginning of the code, a license check page is shown where the user has to input the correct license code in order to proceed and be able to use this code.
22. Finally, the main() function, creates all the additional variable and uses printf() to create the looks of the entire program and using concepts of data abstraction displays only the required information to the user and acts as a driver function to control all the functions created above for their respective job and display all the required details whenever necessary.
23. The main function hence alone acts as the essence of the code and does the complete evaluation of the functions and relevant ordering of function to give a meaningful output.

STOP

Working of Code

The code works in two ways:

First, it takes an infix expression and converts it to postfix using stack (made using linked lists), and displays all possible cases that can exist for this expression for different values of input using a truth table. It the displays the output waveform depending on the clock time interval and number of cycles entered by the user. It also shows the success cases from the truth table, where the value of the output case is 1. It finally, saves the waveform in a file for future reference.

Second, an infix expression is input and the user is asked what each variable means. The user is then asked as to what the output expression is supposed to imply. Then again, the expression is converted to postfix, evaluated and the truth table is displayed. The waveform for the output expression and the success cases are displayed. Also, using the definitions provided by the user we show as to when the output definition is correct and will happen. Now this information i.e. the waveform is saved into a file for future reference.

Conclusion

Therefore, this project tires to encompass and assimilate the integrated concepts of computer science in terms of logical evaluation and C programming. The program can serve as a digital calculator and works on problems entered by users.

This idea can be seen as a prototype that can be developed further into software that can design and analyze circuits and expressions.

Space Complexity: O(n) - Because stack using linked list takes as much space required dynamically and most space is used by evaluation of expression from infix to postfix.

Time Complexity: $O(n^2)$ – Because infix to postfix conversion used two loops to evaluate the expression using stack.

References

www.stackoverflow.com

www.geeksforgeeks.com

www.sanfoundry.com