Practical 4: Logic

Overview

The main object of this practical is to create a program fulfilling the functionalities of a logic solver, that creates and prints truth table for logical formulas. This can be used to prove boolean algebras and to solve logic puzzles.

Format of files (see Makefile for more detail) – ttable executable created using ttable.c, table_io.c, table_io.h. - for extension, ttable2 executable created using ttable2.c, ttable2_io.c, ttable2_io.h and functions in table_io.c

Design & Implementation

There are number of steps the programs takes from accepting inputs to generating outputs.

- Inputs are received by command line arguments and checked in two stages. Whether or not they are in the right format is validated by checking number of arguments, its order, having an integer value within range. Since they are stored in char, function *strtol()* is used to convert it to an int. Secondly, it is checked whether formula is in a correct form, else it is rejected; this is done by *check_form()* function. Every character is checked, using functions such as *isalpha()* and *isdigit()* to identify its property. Number of operands and operators are counted and so in exceptional cases, formula is rejected.

example	a	c	#	1	&	-	a	-	0	|	b	=	-	>
num_var	1	2	1	2	1	1	2	2	3	2	3	2	2	1

This is how *check_form()* checks if the input is the valid formula. *Num_var* increments whenever faced with operands, then if it is two argument operator, *num_var* decrements, whereas if it is one argument operator, it stays the same. Hence, after reaching the end of the formula, *num_var* should be 1. Secondly, I have used the fact that for a single alphabetical character ‘x’, ‘x’ - ‘a’ gives a numeral, in order to recognise variable operands.

- Operator is checked whether it is valid and rejected if unregistered symbol has been used. For computations, bitwise operators such as and, or, exclusive or are used.
- Truth table has been implemented as a struct, consisting of fields such as total number of inputs, inputs, the results of sub-expressions and the final result. Tables are dynamically allocated using and *malloc()* and freed later. Arrays are stored as pointers for efficiency. This helped to isolate the computation and output prints. A separate array containing the overall result is stored, despite that it is the repeat of the last result, considering the cases where the formula only consists of operands.

- Given n, the number of variables to be used for the formula, every binary combinations are generated using the trick of converting 1 to 2^n to binary. For every binary representation, it is then passed onto the function *compute()* to apply the inputs to the formula. Function *get_comb()* will loop around 2^n times to get binary inputs using *get_bin()*, sending as an argument for *compute()*.

- *Compute()* function fulfils the main functionality of applying inputs to get the solutions. It receives one input combination at a time. In order to deal with postfix notation of the formula, stack has been utilised. Stack is represented using a struct, and its necessary operations such as push, pop, top have been written as functions. Hence, while going through every character of the formula string, if it is an operand, that is represented as an alphabetical character, we get the numeral ‘x’ -’a’ using this, and use it as an index to retrieve input value. Then, the operand is pushed onto the stack and in the table it is denoted as -1; this is to print empty string for operands. As it encounters an operator, operand(s) are popped from the stack, applied and result is stored in the created table as a result of sub-expressions.
- As all the results are collected, this is then printed using *print_head()* and *print_table()* functions.

Testing

Stacscheck, Laws of Boolean Algebra - Testing has been done initially using stacscheck which helped to get truth tables printed correctly. Then the program is used to prove the laws of boolean algebra.[1] Since, for each law, two formula have produced the same truth table meaning they are equivalent.

Encoding and Solving Logic Puzzles – for all three puzzles, all statements, variables used for propositional logics, assumptions and outputs are all recorded in **output.txt**. They are all described in detail. The answers are checked by solving the puzzles first by hand and comparing it with the outputs of the truth table.

Additionally, results of solving the complex puzzle from extension is also included in **output.txt** file.

Extensions

For extensions, all five suggestions are implemented in addition to the complex puzzle.

As explained in the specification, these operations are done in ttable2 executable file and each is controlled by the flags; all the combinations of inputs which can be accepted in ttable2 are shown as follows:

n : number of variables, f1 : formula1, f2: formula2

-ps : *Report the status of a formula*

-pt: Prints nothing until a true line is found and then reports it or states that no true line exists.

-dbs: Allow two input strings and print out only the lines where the first one is true.

-ce: Allow two input strings and report whether or not they are equivalent

Number of argument (argc)	Number of flags	Input format		Number of input strings
3	0	./ttable2 n f1 (same as ttable)		1
4	1	./ttable2 -ps n f1	./ttable2 -pt n f1	1
5	1	./ttable2 -dbs n f1 f2	./ttable2 -ce n f1 f2	2
5	2	./ttable2 -ps -pt n f1		1
6	2	./ttable2 -ce -ps n f1 f2	./ttable2 -ce -pt n f1 f2	2
7	3	./ttable2 -ce -ps -pt n f1 f2		2

Implement Dijkstra's Shunting Algorithm (or something similar) to convert input expressions from algebraic notation to RPN.

- for extension, Dijkstra's shunting algorithm is implemented. In the above table, from the second row onwards, if any flag is added for extension, it will automatically perform the shunting algorithm to convert any valid input formulas which are in infix notations to RPN and continue.

- Hence, for proving laws of boolean algebra, we can use *./ttable2 -ce n f1 f2* form to check that they are equivalent.
- Examples of using above input formats are all stored in **ttable2_output.txt**.

Design&Implementation

- Checking correct input format has been done using two functions. Firstly *find_flags* () checks flags and sets corresponding values of f1, f2 and f3. This is then used in *check()* function which considers value of argc, number of flags and flags detected to reject any invalid inputs.

- for shunting algorithm, similarly stack has been used to convert. This time, operators should be stored. Since the struct stack has a pointer to an int array, double amount of work was done to

1 Inputs and outputs have been recorded in output.txt file.

convert operator to relevant value assigned according to the operator precedence and returned back to the symbol character for the overall printing of the resulted formula. These are done by *check_op()* and *get_op()* functions. Alternatively, this could have been improved by storing an union with in struct stack containing int* and char* for uses – for future.
- operator precedence are derived from the c bitwise operator precedence. Brackets are prioritised over operators accordingly.

- in order to check whether a valid infix notation has been given, it is firstly converted to RPN notation then it is checked using *check_form()* function. This is because it is much easier check if it is in postfix notation. I am confident with my shunting algorithm function which allowed me to do this way.

Conclusion

In this practical, I have used various C programming functions, techniques learned thoroughly and extensively from control flow statements, preprocessors, structures, enum types, and memory allocation etc. I have also really enjoyed the practical overall, intrigued by logic solver to solve various puzzles.