**Overview**

In this first CS2002 Practical, I was asked to implement a program to print truth tables for logical formulas in C and use that system to prove some laws of Boolean algebra and solve some logic puzzles. I also had to implement test files to make sure that my implementation worked and followed the format specified by the Practical’s [system specifications](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W05-Logic/CS2002-2024-Logic-Prac.pdf).

I’m proud to say that I’ve achieved the previously established goals:

* I’ve implemented a program in C that takes a logical formula in Reverse Polish Notation (RPN) and prints out its truth table.
* I’ve written unit tests for all my methods.
* I’ve used my implementation to answer the logic questions in Part 2.

**Design and Implementation**

The core of my submission consists of 5 files:

* stack.c and its header file stack.h: my implementation of a stack in C
* print\_table.c and its header file print\_table.h: the “main” method of my submission that prints out the truth table of an imputed logic formula in RPN
* ttable.c: this file reads the command line arguments and calls the *printTable* method in print\_table.c to print out the truth table. This file will be compiled into an executable file to run the program.

I’ve decided to split the print\_table.c and ttable.c files in two to make the testing easier: as print\_table.c does not contain a *main* method, I can just call *printTable* in my test files and compare the output with an expected one. My submission also has a Makefile, which contains different rules to compile and run my code, the testing, and the Part 2 questions: the design and functionality of the Makefile is covered in the README file.

When I first read the Practical’s [system specifications](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W05-Logic/CS2002-2024-Logic-Prac.pdf) and saw that the inputted formula was in RPN, my mind immediately associated it to stacks. Therefore, I decided to use a stack to store the logic formula.

My implementation of a stack ADT, stack.c, contains 7 methods and one *printStack* function. They are all defined in the header file, stack.h. These methods are the typical stack methods:

* Two Boolean methods, *isFull* and *isEmpty*, that return True if the stack is full or empty respectively, and False if not.
* A *getSize* method that returns the number of elements in the stack.
* The trio of *push*, *pop* and *top*: *push* pushes an element to the top of the stack if the stack is not full; *pop* pops the topmost element of the stack if the stack isn’t empty; *top* accesses the topmost element of the stack if the stack isn’t empty WITHOUT removing it.
* A *clear* method that clears the stack.

The stack.c file has one macro variable, MAX, which is defined at 1000: this is the stack’s maximum capacity. According to the Practical’s [system specifications](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W05-Logic/CS2002-2024-Logic-Prac.pdf), “[my] program should accept input strings up to 1000 characters (inclusive)”, and as I used stacks to store this inputted logic formula, my stack cannot take more than 1000 elements.

To store the elements of the stack, I used an array of *char* objects, *items*, as the inputted logical formula is a string of characters. The number of elements in the stack, also used to access the topmost element of the stack, is stored in an integer called *count*, initialised at 0.

The type of my *isFull* and *isEmpty* methods is *int*: the typical True/False values are represented as 1 and 0 respectively in C, therefore the return type of these methods must be *int*. Note that the return type of *getSize* is also *int*, and that it can also return 0 if the stack is empty, or 1 if the stack has one element. However, these integers are not to be interpreted as Boolean values.

My *push* method takes a *char* object as parameter, and, if the stack isn’t full (if *isFull* returns 0), replaces the element of index *count* in *items* with it before increasing *count* by one. For example, if the stack has 35 elements in it, and we want to push a new element to it, it will be the 36th element, so its index in *items* is , therefore we replace the value of *items[count]* with the new element, and we then increase *count* by one. If the stack is full, an error message will be printed, and no elements will be pushed to the stack.

My *pop* and *top* functions are very similar: they both return the topmost element of the stack if the stack is not empty. The only difference is that *pop* removes the element from the stack, whereas *top* only returns its value while keeping it. As the topmost element of the stack has index , *pop* reduces the value of *count* by one before returning the *items[count]* element, whereas *top* only returns the *items[count-1]* element. If the stack is empty, instead of not returning anything, both methods return a ‘?’ character. As this *char* is neither a Boolean variable, a constant, a ‘-‘ sign, or a compound logic operator according to the Practical’s [system specifications](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W05-Logic/CS2002-2024-Logic-Prac.pdf), it is used throughout my implementation as the “error” character: if a ‘?’ is detected in my *printTable* method in print\_table.c, an error message will be printed out and *printTable* will return 1.

To clear the stack, I simply had to set *count* to 0. To print the stack, I cycled through the elements in *items* and printed them with a space. The last element to be printed is the top of the stack. For example, the stack [‘a’, ‘b’, ‘c’, ‘d’, ‘e’] with ‘e’ as the top of the stack will be printed as: “Stack: a b c d e “. The *printStack* method is not used in my implementation of the truth table: it is only used for testing purposes.

The print\_table.c file has 9 different methods that are all defined in its header file, print\_table.h. These methods are:

* The core of my submission, *printTable*. It takes three parameters (the number of command line arguments, the number of Boolean variables, and a string representing the logical formula in RPN), and either prints out the appropriate truth table and returns 0, or an error message and returns 1.
* Two “helper” methods, *getLength* and *power*. *getLength* takes a string as parameter and returns its size, and *power* takes two integers *n* and *exp* and returns . As the Practical’s [system specifications](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W05-Logic/CS2002-2024-Logic-Prac.pdf) states that “[my implementation] should not use any library except the the C standard library”, I assumed that the use of the *string.h* and *math.h* libraries was forbidden, so I had to redefine those methods myself.
* A *getBin* method that converts an integer in base 10 to a binary number in base 2 on a certain number of bits.
* An *insertValue* function that inserts a character in its appropriate place.
* A *getValue* method that takes a character and returns its corresponding value (0 or 1) if it’s a Boolean variable. This method is very important as it also converts negated formulas into atomic formulas (it deals with ‘-‘ signs).
* Two Boolean methods, *isOperator* and *isValidChar*. *isOperator* checks that an inputted character is a valid compound logic operator, and *isValidChar* checks that an inputted character is either a valid Boolean variable, a constant, a ‘-‘ sign, or a compound logic operator.
* The second core method, *compute*, which takes two Boolean variables and one compound logic operator, and returns the value of the compound formula, or the error character ‘?’ if the formula is invalid.

print\_table.c also uses the previously described stack implementation; therefore, it also includes stack.h.

*print\_table* starts by checking for errors in the user inputs: an invalid number of command line arguments, an invalid number of Boolean variables, an invalid size of the formula, or an invalid start of the formula. The Practical’s [system specifications](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W05-Logic/CS2002-2024-Logic-Prac.pdf) state that the executable should take only 2 command line arguments:

* The number of Boolean variables, between 1 and 26.
* The formula in RPN, up to 1000 characters.

In C, the command line arguments include the name of the program. Therefore, the valid number of command line arguments is 3. This number is taken as parameter for the *printTable* method as *agrc*: if *argc* is different than 3, an error message is printed, and the method returns 0. My function also checks that the number of Boolean variables is between 1 and 26: this number, inputted by the used, is stored as *nb\_var* and taken as parameter. If *nb\_var* is smaller than 1 or greater than 26, an error message is printed, and the method returns 0. I then get the inputted formula’s length using my *getLength* method and store it in the *length* variable. If *length* is smaller than 1 or greater than 1000, once again, an error message is printed, and the method returns 0. Finally, for the inputted formula to be in a valid RPN, it must start with either a Boolean variable or a constant. Therefore, if the first element of the formula is an operator or a ‘-‘ sign, an error message is printed, and the method returns 0.

I then created an array of *char* objects, called *variables*, that contains the first *nb\_var* letters of the alphabet. The Practical’s [system specifications](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W05-Logic/CS2002-2024-Logic-Prac.pdf) explains that “the variables used are the first *n* letters of the alphabet”, *n* being *nb\_var*. Therefore, any letter in the formula that isn’t in *variables* will be considered undefined, resulting in an error.

The next step is to create the header of the truth table. Following the Practical’s format, the header contains at maximum 26 Boolean variables, each followed by a space, then a “: “, then the formula of 1000 characters, and finally “ : Result” followed by the empty character ‘\0’. It is thus stored in a string of maximum size . I also created an integer *j* that stores the actual length of the header and another integer *occ­\_var* that counts the number of occurrences of either Boolean variables or constants (both initialised at 0). I then cycled through my *variables* array, and added each letter followed by a space to the header (increasing the value of *j* by one every time). I then added the “: “.

I took adding the formula to the header as an opportunity to verify some aspects of its validity. I cycled through every character in the formula and made sure that it is a valid character, using the *isValidChar* method. If the character is valid, I then check if it’s either a Boolean variable or a constant: if so, I increase the value of *occ\_var* by one. Either way, I add the character to the header and increased the value of *j*. If the character isn’t valid, the method returns 1 (the error message being already printed in *isValidChar*). After adding the formula to the header, I checked that *occ\_var* is greater than 0: if not, the formula does not contain at least one Boolean variable or constant, thus making it invalid. An error message is then printed, and the method returns 1. Finally, I added “ : Result” to the header (while increasing the value of *j*), and printed it out. I then printed a line of *j* ‘=’ symbols underneath it.

The remaining task was to print out the remaining lines of the truth table. While looking at the given example in the Practical’s [system specifications](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W05-Logic/CS2002-2024-Logic-Prac.pdf), I noticed two very important things:

* The number of lines in the truth table (excluding the header and the ‘=’ line) is equal to . So, for 3 Boolean variables, there would be 8 lines.
* If I gave these lines an index from 0 to , the value of the ith Boolean variable in the jth line is equal to the th bit of the index of the jth line in binary on *nb\_var* bits. For example, for 3 Boolean variables, the 4th line would have an index of 3. 3 in binary on 3 bits is 011, therefore the first variable ‘a’ has value 0, the second variable ‘b’ has value 1, and the third variable ‘c’ has value 1.

I created an array of *char* called *bin* to store the value of an integer in binary, and I looped from i = 0 to (for every line in the truth table). I obtained thanks to my *power* method.

In this loop, I started by clearing the stack and pushing every element of the formula to the stack IN REVERSE ORDER: the first element of the formula is the top of the stack. I then updated the *bin* array using my *getBin* method, parsing i as the number to convert on *nb\_var* bits: *bin* now contains the line’s index in binary. To print out the line, I created a string *answer*, which I immediately updated by adding each Boolean variable’s value followed by a space, and the “: “. I then had to determine where the values should be printed in the answer: to do so, I cycled through each character of the formula, checked if it is either a compound logic operator or a ‘-‘ sign, and if so, indicated that location in the *answer* with an ‘@’ symbol, or a space otherwise. At this point, for the formula “ac#1&-a-0|b=->” with 3 variables, the table would be the following:

A screenshot of a computer

Description automatically generated

*Figure 1: Truth table with ‘@’ symbols*

I then had to replace the ‘@’ symbols with the actual values. To do so, I looped while the stack still had more than 2 elements in it, and if there are 2 or more Boolean variables or constants in the formula. I consider the other cases special, and deal with them later. In the *while* loop, I created 4 new variables: an integer *c*, initialised at 0, which is the number of “extra” variables, an array of *char* objects *extra* to store the “extra” variables, and two *char* variables *var1* and *var2*. I pop the top two elements of the stack, call the *getValue* method on them and store the result in *var1* and *var2* respectively. Note that by calling the *getValue* method, the leftmost ‘@’ symbol is replaced by the appropriate value (see *getValue* description further on). I then checked that *var2* is not an operator, as if it’s the case, then the formula is not valid, an error message is printed out and the method returns 1. If not, then I popped the topmost element of the stack, called the *getValue* on it, and stored the returned *char* in a new variable *operator*.

I also must deal with “brackets” in the logical formula: in a formula in RPN, brackets are represented by consecutive variables, followed by consecutive operators. For example, the formula “abc|&” in RPN is equivalent to “(a|b)&c”. This is where “extra” variables come in, as we might have more than two consecutive variables before popping an operator. And the

popped operator only applies to the last two variables. Therefore, while *operator* is not a compound logic operator, I first checked if any of *var1*, *var2* or *operator* is the ‘?’ error symbol. If so, then the formula is missing a compound logic operator, an error message is printed, and the method returns 1. Otherwise, I added *var1* to the *extra* array and incremented *c*, set *var1* to *var2*, *var2* to *operator* and *operator* to a new popped value.

After this *while* loop, both *var1* and *var2* contain either Boolean variables or constants, and *operator* a compound logic operator. I then called the *compute* method to calculate the value of the compound formula, added this result to the *answer* string using *insertValue*, and pushed the result back in the stack. If any “extra” variables were popped, they are pushed back in the stack. At this point, after the *while* loop, for the formula “ac#1&-a-0|b=->” with 3 variables, the table would be the following:

A screenshot of a computer

Description automatically generated

*Figure 2: Truth table after while loop*

We now have two elements or less in the stack. I then popped the topmost element of the stack and called the *getValue* method on it. If the stack is now empty, then the formula was valid, the previously popped element is the result of the formula, so we just push it back. If not, then there are too many Boolean variables/operators, so an error is printed, and the method returns 1. We now have to print *answer*, followed by “ : “, followed by the last element of the stack and “ “. This concludes the *printTable* method, and, for the formula “ac#1&-a-0|b=->” with 3 variables, the table would be the following:

A screenshot of a computer

Description automatically generated

*Figure 3: Truth table after printTable*

The *getLength* and *power* methods are relatively straightforward. *getLength* takes a string as parameter, cycles through every character of the string and increments a counter at every iteration. The function returns the counter value – 1, to not count the ‘\0’ null *char* at the end of the string. *Power* takes two integers, *n* and *exp*, as parameters, and return . Note that my method does not work for negative exponents: I could’ve implemented it, but as I never called the *power* method with negative exponents in my *printTable* method, I decided not to.

*getBin* takes two integers, *dec* (the decimal number) and *nb\_bits* (the number of bits to code *dec* in base 2), and a string *binary*. It converts *bin* to binary on *nb\_bits* bits and updates the *binary* string to store the value. To convert a decimal number to binary, I had to do the Euclidian division of the number by two, until it reaches 0, and store the remainder of each division in reverse order to make the binary number. For example, the process to convert 10 to binary is: remainder 0, remainder 1, remainder 0, remainder 1. So, 10 in binary is 1010.

The *insertValue* method takes a *value* and an *answer* string and replaces the first occurrence of an ‘@’ symbol in *answer* with the *value*. The method loops through each character in *answer* until it reaches an ‘@’ symbol. It then replaces it by *value*. If no ‘@’ symbols were found, the method does nothing (this never happens).

*getValue* is a key method in my implementation. Not only does it return the value (0 or 1) of an inputted Boolean variable and inserts it in the answer string, but it also transforms negated formulas into atomic formula thus dealing with ‘-‘ signs. *getValue* takes three parameters: the character *variable* that I want to get the value of, the string *binary* containing the variable’s values, and the *answer* string. Note that in *printTable*, this function is called with all types of variables (Boolean variables, constants, and compound logic operators). Therefore, I created a new *char* object called *value* that is initialised to *variable*, and this *value* will eventually be modified (or not) in this method. The first thing to check is if the *variable* is a letter, a Boolean operator. This is done by comparing the ASCII value of the character to the value of ‘a’ and ‘z’: if *variable* is greater or equal to ‘a’ and smaller or equal to ‘z’ (in ASCII), then *variable* is a letter. I then set the *value* to the corresponding value of the Boolean variable in *binary*. Knowing that ‘a’ takes the value stored in the first character of *binary*, ‘b’ in the second, … and that the ASCII value of ‘a’ is 97, I deduced that the index of the value of a Boolean variable is stored in the character of index , where *v* is the value of the letter in ASCII. At this point, *value* either is a ‘0’ or a ‘1’ if *variable* is a Boolean variable or a constant. We now must deal with ‘-‘ signs. If *value* is either ‘0’ or ‘1’, and the top of the stack is ‘-‘, then set *value* to ‘1’ if it was ‘0’, and vice-versa, pop the ‘-‘ out of the stack, insert the value in the *answer* using *insertValue* (the value inserted will replace the ‘@’ under the ‘-‘). We have now dealt with variables followed by ‘-‘ signs, but not with variables followed by multiple ‘-‘ signs (e.g. ‘a---‘). Therefore, I call *getValue* once again, but using the updated *value* as my *variable* parameter. If a new ‘-‘ is detected, it will be dealt with the same way as before; if not, then the method returns *value*.

*isOperator* takes a *char* object as parameter and returns 1 if it is a compound logic operator (‘&’ for and, ‘|’ for or, ‘#’ for xor, ‘>’ for implication, and ‘=’ for equivalent), 0 if not. *isValidChar* also takes a *char* object as parameter, as well as an array of defined Boolean variables *variables*, and returns 1 if the character is either a valid Boolean variable (defined in *variables*), a constant, a ‘-‘ sign, or a compound logic operator; 0 if not (as well as an error message).

The last method in *print\_table.c* is *compute*. It takes two variables *var1* and *var2* and a compound logic operator *operator* as parameters and computes the formula ‘*var1* *operator* *var2*’. This function either returns ‘0’ or ‘1’, or the error character ‘?’.

*ttable.c* contains the *main* method used to run the code. It gets the command line arguments, using the *int agrc* and *char \*argv[]* parameters, and calls the *printTable* method (with *argc* as number of command line arguments, *atoi(argv[1])* as the number of variables, and *argv[2]* as the formula). As this file uses methods defined in *print\_table.c*, the header *print\_table.h* is included.

**Testing**

To test my implementation of my code, I created 3 test files:

* *test\_invalid.c*, which tests that my implementation deals with handling errors.
* *test\_stack.c*, which tests my implementation of a stack.
* *test\_print.c*, which tests the functions in *print\_table.c*.

The format of my testing is identical in each of the three testing files. Each test calls the method that it must test and compares the obtained output to an expected value. If they differ, the test fails, the function is considered invalid, both submitted and expected outputs are printed, and the test method returns 1. Otherwise, the test is considered successful, and the test method returns 0.

The detail of the test is printed out when ran: it explains which file is being tested, which method in the file, and in which scenario/for which inputs. To run the tests, simply check the README file for compilation and execution instructions. If my implementation is valid, all 8 stacscheck tests and my unit tests should pass.

**Part 2 Logic Questions**

Question 1:

To prove De Morgan’s law, I converted the formula in RPN and ran my *ttable* executable file with it. If the law is valid, then the inputted formula should be a tautology. I converted De Morgan’s law to “ab|cd||-a-b-&c-d-&&=” and executed *ttable* with 4 variables and this formula. The truth table obtained is the following:

A screen shot of a computer

Description automatically generated

*Figure 4: Truth table for De Morgan’s law*

All result entries are 1, therefore the inputted formula is a tautology, and De Morgan’s law holds.

Question 2:

The question can be interpreted in two different ways. On one hand, let’s suppose that there must be one and only one winner. If so, then we have two possible scenarios:

* If the coin lands on heads, Chris wins
* If the coin lands on tails, Ian loses, therefore Chris wins

In this case, there is only one outcome to the game: Chris wins.

On the other hand, let’s suppose that there can be multiple winners (or even no winners). If so, we have 4 Boolean variables:

* a = “The coin lands on head”
* b = “The coin lands on tails”
* c = “Chris wins”
* d = “Ian wins”

We can also translate the statements in the question into 3 compound formulas:

* “The coin is either heads or tails (but not both)” = “”
* “If the coin lands on heads, Chris wins” = “”
* “If the coin lands on tails, Ian loses” = “”

By combining all three compound formulas into one, and converting them to RPN, we obtain the following: “ab#ac>&bd->&”. If we execute the *./ttable 4 “ab#ac>&bd->&”* command, we obtain the following truth table:

A screen shot of a computer screen

Description automatically generated

*Figure 5: Truth table for question 2*

This formula returns 1 when:

* a = 0, b = 1, c = 0, d = 0: The coin lands on tails, so Ian loses, and Chris loses.
* a = 0, b = 1, c = 1, d = 0: The coin lands on tails, so Ian loses, and Chris wins.
* a = 1, b = 0, c = 1, d = 0: The coin lands on heads, so Chris wins, and Ian loses.
* a = 1, b = 0, c = 1, d = 1: The coin lands on tails, so Chris wins, and Ian wins.

Question 3:

We have 5 Boolean variables:

* a = “Ann attended the dinner”
* b = “Barbara attended the dinner”
* c = “Charles attended the dinner”
* d = “Deborah attended the dinner”
* e = “Elanor attended the dinner”

We can also translate the facts in the question into 5 compound formulas:

* “Either Deborah or Charles or both attended the dinner” = “”
* “Either Barbara or Eleanor but not both attended” = “”
* “If Ann attended, then so did Barbara” = “”
* “Eleanor attended if and only if Deborah attended” = “”
* “If Charles attended, then both Ann and Deborah attended” = “”

By combining all five compound formulas into one, and converting them to RPN, we obtain the following: “dc|be#&ab>&ed=&cad&>&”. If we execute the *./ttable 5 “*dc|be#&ab>&ed=&cad&>&*”* command, we obtain the following truth table:

A screenshot of a computer screen

Description automatically generated

*Figure 6: Truth table for question 3*

This formula returns 1 only when a = 0, b = 0, c = 0, d = 1 and e = 1: only Deborah and Elanor attended the dinner.

Question 4:

We have 9 Boolean variables:

* a = “Box 1 contains the red card”
* b = “Box 1 contains the black card”
* c = “Box 1 contains the prize”
* d = “Box 2 contains the red card”
* e = “Box 2 contains the black card”
* f = “Box 2 contains the prize”
* g = “Box 3 contains the red card”
* h = “Box 3 contains the black card”
* i = “Box 3 contains the prize”

We can also translate the statements in the question into 8 compound formulas:

* “One box contains either a red card, a black card, or a prize but no cards” = “” = “” = “”
* “One box contains a red card, one a black card, and the other contains a prize but no cards” = “” = “” = “”
* “Box 1: This box contains a prize” is True = “”
* “Box 1: This box contains a prize” is False = “”
* “Box 2: The sentence on Box 1 is true” is True = “”
* “Box 2: The sentence on Box 1 is true” is False = “”
* “Box 3: Box 2 contains a black card” is True = “”
* “Box 3: Box 2 contains a black card” is False = “”

By combining all eight compound formulas into one, and converting them to RPN, we obtain the following: “ab#c#ab&c&#de#f#de&f&#&gh#i#gh&i&#&ad#g#ad&g&#be#h#be&h&#&cf#i#cf&i&#&&ac>&bc->&dc>&ec->&ge>&he->&”. If we execute the *./ttable 9 “ab#c#ab&c&#de#f#de&f&#&gh#i#gh&i&#&ad#g#ad&g&#be#h#be&h&#&cf#i#cf&i&#&&ac>&bc->&dc>&ec->&ge>&he->&”* command, we obtain a truth table, and here’s the header and an extract:





*Figures 7 and 8: Header and extract of the truth table for question 4*

This formula returns 1 only for c = 1, d = 1, h = 1, and the rest of the variables are 0: Box 1 has the prize, Box 2 has the red card, and Box 3 has the black card.

**Conclusion**

All in all, I have found this practical quite challenging, as this was the first time (excluding the W02-Exercise) coding in C. I was not used to pointers and memory allocations, and I struggled with formatting my truth table. However, after implementing unit tests and reviewing the examples give in lectures, I managed to implement the required truth table generator.

If I had extra time, I would try to implement a more memory-efficient stack implementation, as mine always has a maximum capacity of 1000, regardless of the size of the inputted formula. Also, for 26 Boolean variables and the formula “*abcdefghijklmnopqrstuvwxyz&&&&&&&&&&&&&&&&&&&&&&&&&*”, my implementation takes approximately 12 minutes to run, which I believe isn’t the most efficient implementation in terms of speed.