

Overview

In this practical, we were tasked with developing a C program that would be able to build truth tables for logical formulas written in reverse polish notation. After implementing the program, we were tasked with using it to solve 4 logical problems.

Part 1 – the ttable program

Design

Checking the arguments

In order to ensure that the arguments obey the constraints listed in the specification, the program checks the following things:

- That the correct number of arguments was supplied
- That the number of variables is in range (between 1 and 26)
- That the formula is 1000 chars or shorter

To parse the var count argument I have written a wrapper function for `strtol`, which parses the supplied string into a long, checks that the number will not overflow an integer, converts long to int and returns it. If it detects an overflow, it will print an error message and exit with code 1.

Stack

Since we will need some sort of LIFO buffer to parse the formula in RPN, we will implement a stack, as described to us in CS2001.

To accomplish this, we declare a struct `BoolStack`, which stores the size of the stack and the pointer to the first element

To store the values efficiently, we use dynamically memory allocation. This is achieved by using `realloc()`, which allows expanding and shrinking already allocated memory.

When we push an item into the stack, `realloc` will take in the pointer to the beginning of the stack, extend the allocated memory by `sizeof(bool)` and return a new pointer to the stack (If a new memory block has to be allocated, `realloc` automatically deals with moving all the information stored in the stack to the new place)

When we pop an item from the stack, `realloc` will shrink allocated memory by `sizeof(bool)`, freeing the extra bytes that were taken up by the retrieved item.

If `pop()` is called for an empty stack, the function will print an error message and exit with code 1.

Once the program is done with the stack, it should be destroyed using the `destruct()` function, this frees any memory still allocated for the stack.

Parsing the RPN formula

Parsing reverse polish notation is rather simple to implement, since don't have to worry about operation precedence, and therefore can just read and immediately evaluate the formula from left to right. We will use the stack, we implemented earlier, as a buffer for formula's intermediate values.

The evaluation loop works like this:

- Read the next char in the formula (starting from `formula[0]`)

- Declare a bool variable `val`
- Evaluate `val`, based on the current character (formula)
- Push `val` into the buffer
- Repeat

Evaluating val

Each iteration of the loop should finish with `val` equal to the value of the formula processed during this loop. These formulas are processed as follows:

atomic formula

Atomic formulas are constants or variables, they evaluate to themselves

negated formula

If the `currentChar` is equal to the negation operator `'-'` we have encountered a negated formula, in this case we will retrieve a single value from the stack (this will be the value of the formula before the negation) and negate it. This will be our `val` for this iteration.

compound formula

If the `current char` is none of the above, it is assumed to be a compound formula operator. In this case, we will retrieve two values from the stack (which will be the values of the last two formulas) and try to perform a compound operation on them by passing them to `eval_compound()` if the character was indeed a compound operator we will receive the result of the operation and write that into `val` for this iteration. If the character isn't a valid compound operator – `eval_compound()` will return -1, and we will print an error message to the user and exit.

Providing the variable values to print_for_vars() efficiently

Since we want to evaluate the function for all possible variable combinations, we need some way of providing the values of these variables to our evaluation function.

One can easily notice, that if we count to $2^{(\text{number of vars})}$ in binary, we will go through all possible variable value combinations. Thus, if we want to go through every possible variable combination in order, we can simply pass all numbers from 0 to $2^{(\text{number of vars})}$ to `print_for_vars()`, which then can map the Nth bit of the number to the value of the Nth variable.

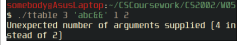
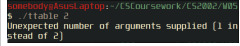
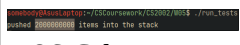

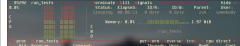
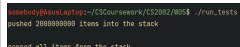
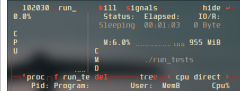
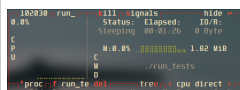
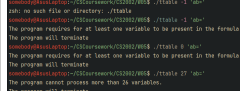
The makefile

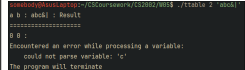
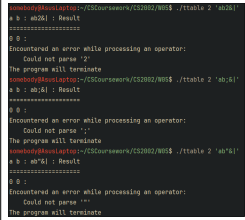

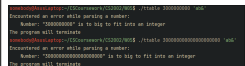
In order to automate building of the program, I have written a makefile. This makefile allows the user to build a `ttable` executable by calling `make ttable` from the project's root or `./src` (the Makefile is symlinked into `./src` for compatibility with `stacscheck`)

The executable and object files will be placed in `./bin` directory, and the source files will be sourced from `./src` directory. This is done using `make`'s support for variables and directory search.

Since all object files are built using the same command, I have created a pattern rule, which allows building a `.o` file from any `.c` file.

Testing

Test Description	Input / Initial conditions	Expected Result	Evaluation/ Comments
StacsCheck tests for - normal operation - formula too long	Run the stacsckeck tests	8 out of 8 passes	All tests passed
Make from scratch	Remove all compiled binaries and ./bin, try to build the program	Ttable is built into project root, .o files are built into ./bin	All built as expected
Oversupply arguments	\$./ttable 3 'abc&&' 1 2	Unexpected number of arguments supplied (4 instead of 2) is printed, the program exits	As expected: 
Undersupply arguments	\$./ttable 2	Unexpected number of arguments supplied (1 instead of 2)	As expected: 
Stress test memory (memory_tests.c)	Push 2000000000 items into the stack	This operation will reserve a considerable amount of memory, but should work flawlessly	Operation successful:  1.86 Gib was reserved 
Check for memory leaks (memory_tests.c)	Pop 2000000000 items we just pushed	The majority of 1.86 Gib should be freed	Most of the memory was freed:  
Check that destruct() works	Populate a stack in a function Destruct the stack and exit the function	The memory allocated to the stack should be freed	Populated the stack:  Destructed the stack 
Check that illegal var counts get rejected	\$./ttable -1 'ab=' \$./ttable 0 'ab=' \$./ttable 27 'ab='	The program should refuse to build a table	As expected: 

Check that illegal var names get rejected:	\$./ttable 2 'abc& '	The program should stop building the table and exit the program as soon as it hits 'c'	As expected: 
Check that illegal symbols get rejected	\$./ttable 2 'ab2& ' \$./ttable 2 'ab;& ' \$./ttable 2 'ab"& '	The program should stop building the table and exit the program as soon as it hits the illegal char	As expected: 
Check that the program exits gracefully when the formula is unsolvable	\$./ttable 2 'ab10& ' ^ to many operands \$./ttable 2 'ab& ' ^ to many operators	An error message should be printed as soon as it becomes apparent that the formula is unsolvable	As expected: 
Check that protection from int overflow works	\$./ttable 3000000000 'ab*& \$./ttable 3000000000000000000000000000 'ab*&	The program should print an error message and exit	As expected: 

Part 2 – solving problems

De Morgan's law proof

To prove that the statement we are provided with is correct, we would have to prove that it is a tautology – true for every combination of inputs. Since this statement has a rather small number of inputs, we could use the program written earlier to build a complete truth table. If all results fields in this table are equal to 1 – that means that the statement holds.

Since the program requires the formula to be written in RPN, we'll have to translate it. To do so, we follow the order of operations denoted by the brackets, and rewrite each expression with the operands followed by the operator.

The translated expression looks like this:

```
ab|cd||-a-b-&c-d-&&=
```

Now we can run the program for this formula:

```
$ ./ttable 4 'ab|cd||-a-b-&c-d-&&='
```

The result is as follows:

```

a b c d : ab|cd||-a-b-&c-d-&&= : Result
=====
0 0 0 0 : 0 001 1 11 1 1111 : 1
0 0 0 1 : 0 110 1 11 1 0001 : 1
0 0 1 0 : 0 110 1 11 0 1001 : 1
0 0 1 1 : 0 110 1 11 0 0001 : 1
0 1 0 0 : 1 010 1 00 1 1101 : 1
0 1 0 1 : 1 110 1 00 1 0001 : 1
0 1 1 0 : 1 110 1 00 0 1001 : 1
0 1 1 1 : 1 110 1 00 0 0001 : 1
1 0 0 0 : 1 010 0 10 1 1101 : 1
1 0 0 1 : 1 110 0 10 1 0001 : 1
1 0 1 0 : 1 110 0 10 0 1001 : 1
1 0 1 1 : 1 110 0 10 0 0001 : 1
1 1 0 0 : 1 010 0 00 1 1101 : 1
1 1 0 1 : 1 110 0 00 1 0001 : 1
1 1 1 0 : 1 110 0 00 0 1001 : 1
1 1 1 1 : 1 110 0 00 0 0001 : 1

```

As we can see, every combination of inputs evaluates to true (1), therefore the law holds.

The coin game

As was discovered during our W04 tutorial, the rules of this game don't define if the game can only have one winner. If we don't make this assumption, we cannot predict the outcome of the game, since we do not know what happens to Ian if Chris wins as well as what happens to Chris if Ian loses.

Therefore, in order to answer the question, we will have to assume that Ian and Chris are playing a zero-sum game (i.e. either Chris or Ian has to win, both can't win at the same time)

For the rules of the game to be upheld, all the following statements should be true:

```
H → C - if coin lands heads, Chris wins
T → ¬I - if coin lands tails, Ian loses (doesn't win)
H ∨ T - the coin can only land on one side
I ∨ C - only one player can win the game (an assumption, see above)
where:
H - coin lands heads, T - coin lands tails, I - Ian wins, C - Chris wins,
∨ - the XOR operator
```

When translated into RPN this statement looks as follows: $ac > bd \rightarrow ab \# cd \# \&\&\&$, where $a = H$, $b = T$, $c = C$, $d = I$.

We can now put this formula into the program, and get a truth table generated.

```
$ ./ttable 4 'ac>bd->ab#cd#&&&'

a b c d : ac>bd->ab#cd#&&& : Result
=====
0 0 0 0 : 1 11 0 0000 : 0
0 0 0 1 : 1 01 0 1000 : 0
0 0 1 0 : 1 11 0 1000 : 0
0 0 1 1 : 1 01 0 0000 : 0
0 1 0 0 : 1 11 1 0000 : 0
0 1 0 1 : 1 00 1 1100 : 0
0 1 1 0 : 1 11 1 1111 : 1
0 1 1 1 : 1 00 1 0000 : 0
1 0 0 0 : 0 11 1 0000 : 0
1 0 0 1 : 0 01 1 1110 : 0
1 0 1 0 : 1 11 1 1111 : 1
1 0 1 1 : 1 01 1 0000 : 0
1 1 0 0 : 0 11 0 0000 : 0
1 1 0 1 : 0 00 0 1000 : 0
1 1 1 0 : 1 11 0 1000 : 0
1 1 1 1 : 1 00 0 0000 : 0
```

As we can see, there are only two cases when the rules of the game are upheld, and in both cases c has to be true and d (I) has to be false. Therefore, this game can end only in Chris winning and Ian losing.

Dinner

To solve this puzzle, we will once again write down logical statements for all the facts. We will then pass them into the program, and find a (hopefully single) combination of inputs that results in all statements being true.

The logical statements are as follows:

```
D ∨ C
B ∨ E
A → B
¬(E ∨ D) <- either both missing or both present, exact opposite of Xor
C → (A ∧ D)
Where each letter signifies if the person was in attendance
```

The final RPN formula looks like this: `dc|be#ab>ed#-cad&>&&&&`

Passing this formula to the program yields:

```
./ttable 5 'dc|be#ab>ed#-cad&>&&&&'

a b c d e : dc|be#ab>ed#-cad&>&&&& : Result
=====
```

0	0	0	0	0	:	0	0	1	01	011100	:	0
0	0	0	0	1	:	0	1	1	10	010000	:	0
0	0	0	1	0	:	1	0	1	10	010000	:	0
0	0	0	1	1	:	1	1	1	01	011111	:	1
0	0	1	0	0	:	1	0	1	01	000000	:	0
0	0	1	0	1	:	1	1	1	10	000000	:	0
0	0	1	1	0	:	1	0	1	10	000000	:	0
0	0	1	1	1	:	1	1	1	01	000000	:	0
0	1	0	0	0	:	0	1	1	01	011110	:	0
0	1	0	0	1	:	0	0	1	10	010000	:	0
0	1	0	1	0	:	1	1	1	10	010000	:	0
0	1	0	1	1	:	1	0	1	01	011100	:	0
0	1	1	0	0	:	1	1	1	01	000000	:	0
0	1	1	0	1	:	1	0	1	10	000000	:	0
0	1	1	1	0	:	1	1	1	10	000000	:	0
0	1	1	1	1	:	1	0	1	01	000000	:	0
1	0	0	0	0	:	0	0	0	01	011000	:	0
1	0	0	0	1	:	0	1	0	10	010000	:	0
1	0	0	1	0	:	1	0	0	10	110000	:	0
1	0	0	1	1	:	1	1	0	01	111000	:	0
1	0	1	0	0	:	1	0	0	01	000000	:	0
1	0	1	0	1	:	1	1	0	10	000000	:	0
1	0	1	1	0	:	1	0	0	10	110000	:	0
1	0	1	1	1	:	1	1	0	01	111000	:	0
1	1	0	0	0	:	0	1	1	01	011110	:	0
1	1	0	0	1	:	0	0	1	10	010000	:	0
1	1	0	1	0	:	1	1	1	10	110000	:	0
1	1	0	1	1	:	1	0	1	01	111100	:	0
1	1	1	0	0	:	1	1	1	01	000000	:	0
1	1	1	0	1	:	1	0	1	10	000000	:	0
1	1	1	1	0	:	1	1	1	10	110000	:	0
1	1	1	1	1	:	1	0	1	01	111100	:	0

The only line that evaluates to true is:

```
0 0 0 1 1 : 1 1 1 01 011111 : 1
```

Therefore, we can state that only Deborah and Eleanor attended

The boxes of Scheherazade

To solve this puzzle, we will start by encoding the rules of the game into propositional logic.

Let a , b , c be true if the sentence on the respective box (1,2,3) is true

Then all of the following must be true for the rules of the game to be upheld

$(a \vee b \vee c)$ – at least one statement has to be true

$\neg(a \wedge b \wedge c)$ – at least one statement has to be false

$b = a$ – According to statement on box 2

$c = \neg b$ – if c is true, then box 2 has a black card, thus b has to be false and
if b is true, then box 2 can't have a black card, thus c is false

This, however, is not enough to be able to definitively tell which box contains the prize.

Fortunately, we can reach a simple logical conclusion from the text of the riddle:

Given that the statement on box 1 says that box 1 has a prize, this statement will be false only if it has a black card. Since statement c informs us of the position of the black card, we can come to a following conclusion: $c \rightarrow a$

Now we can translate those into a formula written in RPN and pass it to the program.

The final formula is:

```
abc | | abc&&-ba=cb-=ca>&&&&
```

Passing this formula to the program yeilds:

```
./bin/ttable 3 'abc | | abc&&-ba=cb-=ca>&&&&'
```

```
a b c : abc | | abc&&-ba=cb-=ca>&&&& : Result
```

0	0	0	:	00	001	1	10	10000	:	0
0	0	1	:	11	001	1	11	00000	:	0
0	1	0	:	11	001	0	01	11000	:	0
0	1	1	:	11	101	0	00	00000	:	0
1	0	0	:	01	001	0	10	10000	:	0
1	0	1	:	11	001	0	11	11000	:	0
1	1	0	:	11	001	1	01	11111	:	1
1	1	1	:	11	110	1	00	10000	:	0

As we can see, the only line, for which all statements hold is $a=1$ $b=1$ $c=0$, thus:

- Box 1 has to have a prize in it, since a true statement a says it's there
- Box 2 has to have a red card in it, since statement b written on it is true
- Box 3 has to have a black card in it, since statement c written on it is false

Evaluation

Overall, this practical was quite a lot of fun to write. I especially enjoyed the open endedness of the specification. My final program complies with all the requirements listed in the spec, and it has helped me with the second part of the practical significantly.

Despite this, I still encountered some challenges while working on this practical.

The most major of those I encountered while working on implementing the stack: The program would segfault if the stack grew larger than 134473 elements.

After unsuccessful debugging efforts, I revisited the page on realloc on cppreference.com and discovered that I had overlooked the fact that realloc does not automatically reassign the pointer to a new memory block. Actually, realloc returns a new pointer and the programmer should handle reassignment themselves.

On discovering that, I adapted the example provided on that page, which checks if the memory reallocation was successful and reassigns the pointer if it was, and the code has worked flawlessly since then.

Conclusion

In conclusion, this practical has allowed me to practice coding in C, as well as solving logical problems using truth tables. Having completed it, I'm more confident in my low-level programming abilities. I had also learned more about writing makefiles, which will certainly come in useful in the following practicals.

References

- <https://en.cppreference.com/w/c> – cppreference's C reference pages, used to find information on C standard library functions and language features.
- <https://www.gnu.org/software/make/manual/make.html> – The GNU Make manual, used to write the makefile. Find links to chapters used in comments inside of the makefile