cexpr: C Expression Solver

# Synopsis

“cexpr”, otherwise known as the “C Expression Solver” is an assignment where we have to use Lex and Yacc to generate a parser that will accept C-like expressions, then compute values from it. It will allow us to use variables (26 actually) based on the lowercase letters of the alphabet.

# Approach

The solution, when you understand Lex and Yacc, is fairly trivial to make. However, I took an approach to perfectly match the solution executable, even in the most brutal of equations. Writing the Yacc code was trivial to assign values to variables, implement left-recursion, etc. And I even modified the Lex code to accept spaces because it got annoying to not do that otherwise.

# Debugging Solution

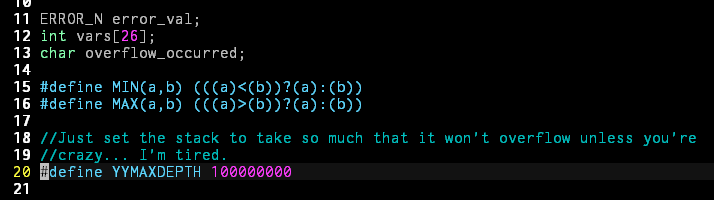
My method of testing my code against the solution was by sheer brute-force. Test as many equations as possible to ensure that it worked properly. I have written some generators in Node.js to generate hundreds of thousands of extremely intensive mathematical equations for both the solution and my own code to run. This is because, once again, my goal is to make my solution match the professor’s solution exactly, in a bit-to-bit basis, even in the most intensive equations possible, and even in undefined behaviour. Brute-forcing by generating hundreds of thousands of equations allowed me to fully test how mine held up to the solution. You may view the next section of this document (“Issues”) to see what methods I actually had to observe in order to force mine to match the solution executable and the issues I encountered. It’s fairly lengthy. I only went this far on the assignment because I wanted a challenge, so I set myself out to make it one.

# Issues

It’s novel time. There were a lot of problems that I encountered in my testing, because the equations generated were so extreme and were in such a high quantity that I was actually surprised that *even the solution executable* held up. It ended up being very humourous at how many issues were encountered. We will start off with the simplest and then go more in depth.

## Yacc stack overflow

One of the most annoying issues I ran into was the Yacc stack overflow. Whenever I generated example equations, the first thing I wanted to test was how many equations the solution executable and my own executable could take. The solution was able to handle well over 1 million entries. But whenever I ran the same equations on my own, it handled around 490 (average of 200 test runs) equations. All of those times ended up with a “Yacc stack overflow” around that threshold. My first guess was to remove all shift/reduce and reduce/reduce conflicts since mine had around 140 of those. I ended up reducing most of the issues myself and got rid of all of the conflictions but still got the “Yacc stack overflow”. I found 2 ways to bypass this error. One was as simple as throwing “yyparse” into a loop so it’d keep running. Then when the parser encountered an error, it would just restart where it left off. This, however, meant that broken equations such as “Aa2a;” did *not* break the parser, as it would just restart. Then I realised the solution was simply in Yacc’s configurations.



Setting “YYMAXDEPTH” value to an absurdly high value allowed me to take on **hundreds of millions** of equations before encountering another stack overflow. I consider that a workaround because I doubt someone would run more than 100,000,000 equations unless they worked for NASA.

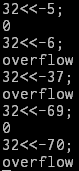
## Bitshifting

### Simple Undefined Behaviour & Reverse Engineering Solution Code

You would think that bitshifting would be simple, and you’re right. However, the solution executable has a few issues with undefined behaviour. Take the following equations into consideration:

32<<-5;

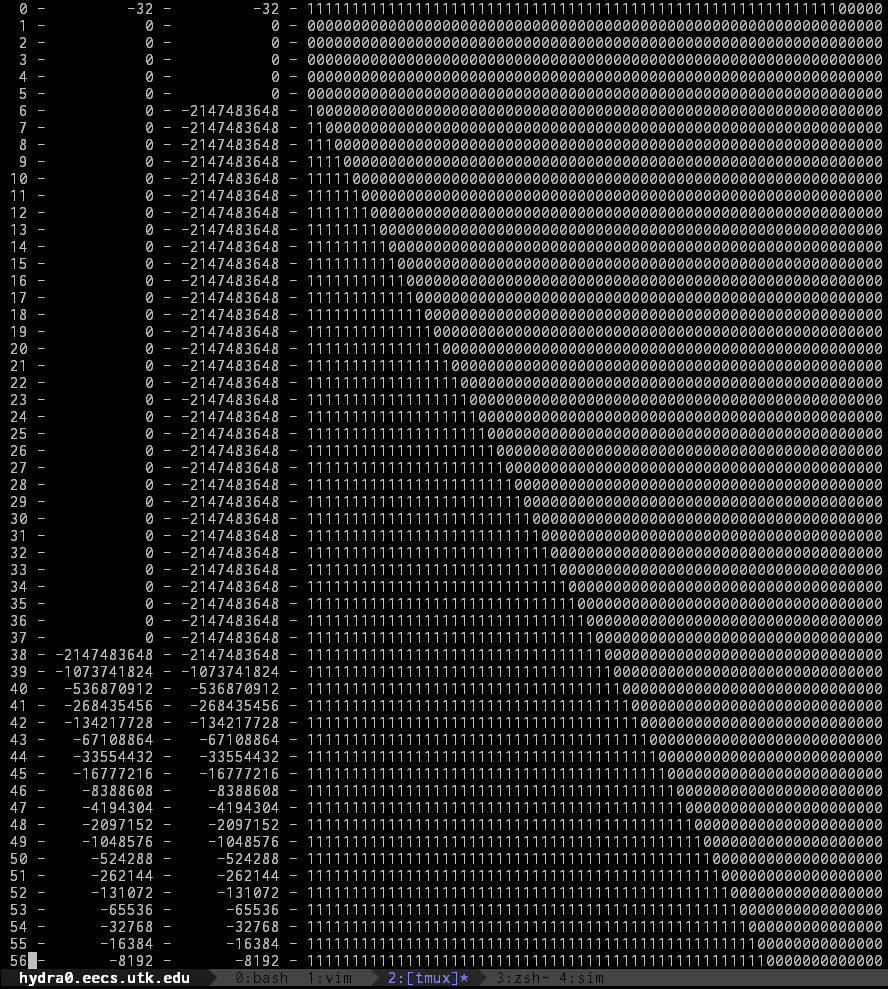
32<<-6;

Both of these are examples of **undefined behaviour** due to bitshifting by a negative value (See “ISO 9899:1999 6.5.7” at the bottom of this document). In my efforts to match the executable exactly, I very quickly realized that these values were not matching. The first one results in “0” while the second results in “overflow”. This forced me to try to break the solution executable and reverse engineer how it worked without the solution code. My first attempt was to see how exactly the solution was storing the integer before doing operations. The simple way was to decrement the bitshift amount of -5 by 32 and 64 to see what would happen. The results were intriguing as shown to the right:

When we took -5 and subtracted 32, we ended up with -37, which overflowed. But when we subtracted by 64, we got what we started with. Upon doing more tests with other numbers, I managed to conclude that the solution executable casted the “integer” into a “long long” or another 64-bit data type. This meant that I had to do the same in order to match the solution code even in undefined behaviour circumstances… right? I tested out my code against his and ended up matching in this situation.

### Negative Numbers being bitshifted

What about -32 being shifted? Also, what about a negative number being bitshifted by a negative value? Turns out that this broke my code as well. Eventually I got pissed off enough to write a test C program to analyse what exactly was going on. Consider the following equation: 32<<-x;



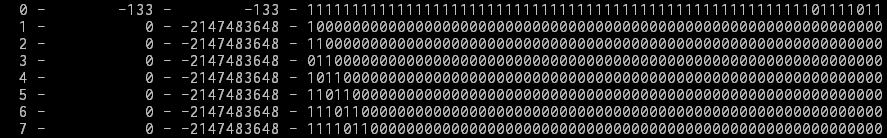
So what exactly is going on? This was generated by “bitshift\_l.c” in the “test” directory of this submission. The way how the solution executable tests for overflow with these kinds of numbers is simple. If the last 32 bits is not 0, then set the 32nd bit (signed bit) to 1. Otherwise, set it to 0. Check if the signed bit has changed from prior to the bitshift. And if it is, then force the output to say that it overflowed.

### An even more extreme case.

Consider the following equation:

-133<<-1409;

This equation resolves to “-2147483648” in gcc compiled C code, in JavaScript, and in my own code. However, it is marked as an “overflow” in the solution code. Of course, trying to shift by a huge number like -1409 is *absurd*. We know that the solution executable stores this as a 64-bit integer, so let’s modulo the absolute value of that number by 64 and get a more appropriate value. We end up with -1 (or 63). Both result in an “overflow” in the solution executable, but not in my code. So time for more bit-level observations.



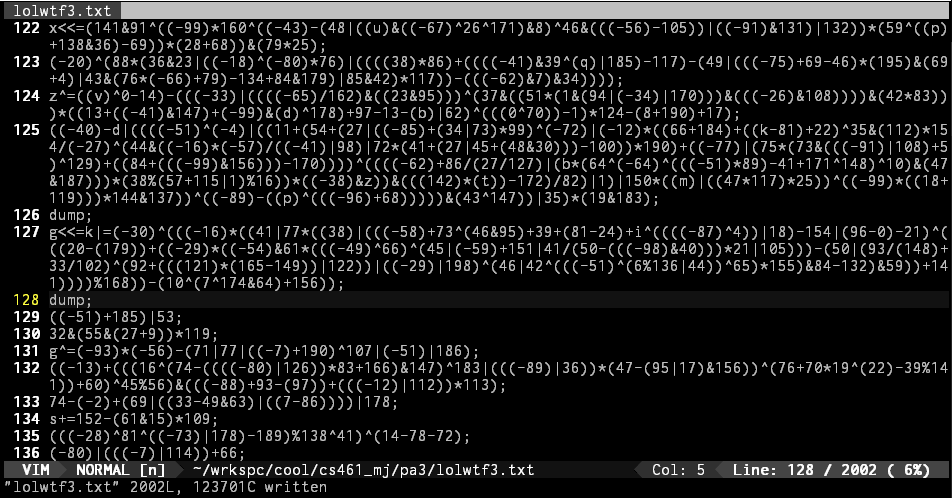
The algorithm discussed in the previous section “Negative Numbers being bitshifted” was affecting this as the number on the left side was negative, as well as the right side. The last 32 bits was not 0. Therefore, by the algorithm in the previous section, it forced the last bit in the casted integer to be negative, resulting in “-2147483648”. The fix was as simple as checking whether or not the result was equal to “0x80000000”. However, it didn’t make the other test cases I made happy. But it definitely allowed me to match the solution code with undefined behaviour. Though I wish it was a lot simpler.

### Final thoughts of reverse engineering

I would think that if I simply made the integer as a “long long”, I could avoid all of this, but I needed something challenging to try, considering the rest was disappointingly trivial. Therefore, *I purposefully made it like this just so I can have some fun with the assignment.* I would like to look at how the solution code was implemented. My “reverse engineering” didn’t go as far as some developers would go to understand how something works (not even close), but I’m glad I got to do some of it for this assignment. The solution code was compiled on the same machine as my code, so I would’ve thought that the undefined behaviour would match just because of that alone. But obviously the implementation was different, so of course I had to compensate for that.

## Testing Methodology

Here is a screenshot of some of the stuff I did in order to brute force and ensure that my solution matched the solution executable’s output:



Forcing the students to implement a “dump” function was very handy in debugging, as it allowed me to find out the issues for a *lot* of my operations while writing this assignment (line 127 is what actually gave me a lot of problems with bitshifting, as shown above).

# Gradescripts

As expected, I have generated 125 gradescript test cases for this lab assignment too, which automates the diff comparisons for the end user. However, since some of the text files are huge (containing well over tens of thousands equations of more than 50 operations each), they had to be compressed using XZ-utils. The gradescript will handle everything. See the next page for the results of the gradescript on my end. The first 100 are “casual”. The next 23 are more brutal. 124 and 125 push the limits of the solution code, as well as the student’s code, to the most extreme level that I could possibly throw at it. They have the most complexity that I could possibly generate.

# Specifications (C Standard)

## ISO 9899:1999 6.5.7

“The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. **If the value of the right operand is negative** or is greater than or equal to the width of the promoted left operand, **the behavior is undefined**.”

As usual, here’s an example of the gradescript working its magic. It was a fun assignment.

