

# 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 humorous 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 conflicts 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.

```
10
11 ERROR_N error_val;
12 int vars[26];
13 char overflow_occurred;
14
15 #define MIN(a,b) (((a)<(b))? (a):(b))
16 #define MAX(a,b) (((a)>(b))? (a):(b))
17
18 //Just set the stack to take so much that it won't overflow unless you're
19 //crazy... I'm tired.
20 #define YYMAXDEPTH 100000000
21
```

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

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

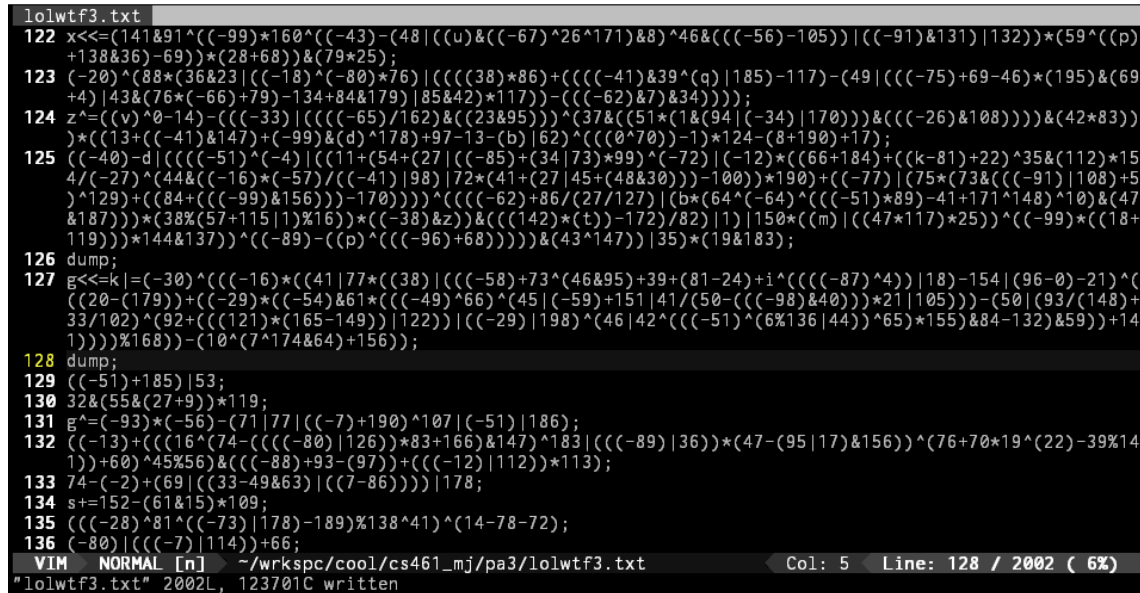
$$32 < -x;$$

So what exactly is going on? This was generated by "bitshift.c" in the "test" directory of this



## 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:



```
lolwtf3.txt
122 x<=<=(141&91^(((-99)*160^((-43)-(48|((u)&((-67)^26^171)&8)^46&((-56)-105))|((-91)&131)|132)))*(59^((p)
+138&36)-69))*(28&68)&(79*25);
123 (-20)^(88*(36&23|((-18)^(-80)*76)|(((38)*86)+((( (-41)&39^((q)|185)-117)-(49|((-75)+69-46)*(195)&(69
+4)|43&(76*(-66)+79)-134+84&179)|85&42)*117))-((( (-62)&7)&34)))));
124 z^=((v)^0-14)-((( (-33)|((( (-65)/162)&(23&95)))^37&((51*(1&(94|(-34)|170)))&((-26)&108))))&(42*83))
)*(13+((( (-41)&147)+((-99)&(d)^178)+97-13-(b)|62)^((0^70))-1)*124-(8+190)+17);
125 ((-40)-d|((( (-51)^(-4)|((11+(54+(27|((-85)+34|73)*99)^(-72)|(-12)*((66+184)+((k-81)+22)^35&(112)*15
4/((-27)^44&((-16)*(-57)/((-41)|98)|72*(41+(27|45+(48&30)))-100))*190)+((-77)|75*(73&((-91)|108)+5
)^129)+((84+((( (-99)&156)))-170)))^((( (-62)+86/(27/127)| (b*(64^(-64)^((( (-51)*89)-41+171^148)^10)&(47
&187)))*(38*(57+115|1)*16))*((-38)&z))&(((142)* (t))-172)/82)|1)|150*( (m)|((47*117)*25))^((-99)*((18+
119)))%144&137))^((-89)-((p)^((( (-96)+68))))&(43^147))|35)*(19&183);
126 dump;
127 g<=<=k|=(-30)^((( (-16)*((41|77*(38)|((-58)+73^46&95)+39+(81-24)+i^((( (-87)^4))|18)-154|(96-0)-21)^((
(20-(-179))+((-29)*((-54)&61*(( (-49)^66)^45|(-59)+151|41/(50-((-98)&40))*21|105)))-(50|(93/(148)+
33/102)^92+(((121)*(165-149))|122))|((-29)|198)^46|42^((( (-51)^6%136|44))^65)*155&84-132)&59)))+14
1)))%168))-((10^7^174&64)+156));
128 dump;
129 ((-51)+185)|53;
130 32&(55&(27+9))*119;
131 g^=(-93)*((-56)-(71|77|((-7)+190)^107|(-51)|186);
132 ((-13)+(((16^74-((( (-80)|126))*83+166)&147)^183|((( (-89)|36))*47-(95|17)&156))^76+70*19^(22)-39%14
1))+60)^45%56&((( (-88)+93-(97))+((-12)|112))*113);
133 74-((-2)+69|((33-49&63)|((-7-86))))|178;
134 s+=152-(61&15)*109;
135 (((-28)^81^((-73)|178)-189)%138^41)^14-78-72);
136 (-80)|((( (-7)|114))+66;
VIM NORMAL [n] ~/wrkspc/cool/cs461_mj/pa3/lolwtf3.txt Col: 5 Line: 128 / 2002 ( 6%)
"lolwtf3.txt" 2002L, 123701C written
```

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.

```
( 72/125) Checking "071.txt"... [PASSED]
( 73/125) Checking "072.txt"... [PASSED]
( 74/125) Checking "073.txt"... [PASSED]
( 75/125) Checking "074.txt"... [PASSED]
( 76/125) Checking "075.txt"... [PASSED]
( 77/125) Checking "076.txt"... [PASSED]
( 78/125) Checking "077.txt"... [PASSED]
( 79/125) Checking "078.txt"... [PASSED]
( 80/125) Checking "079.txt"... [PASSED]
( 81/125) Checking "080.txt"... [PASSED]
( 82/125) Checking "081.txt"... [PASSED]
( 83/125) Checking "082.txt"... [PASSED]
( 84/125) Checking "083.txt"... [PASSED]
( 85/125) Checking "084.txt"... [PASSED]
( 86/125) Checking "085.txt"... [PASSED]
( 87/125) Checking "086.txt"... [PASSED]
( 88/125) Checking "087.txt"... [PASSED]
( 89/125) Checking "088.txt"... [PASSED]
( 90/125) Checking "089.txt"... [PASSED]
( 91/125) Checking "090.txt"... [PASSED]
( 92/125) Checking "091.txt"... [PASSED]
( 93/125) Checking "092.txt"... [PASSED]
( 94/125) Checking "093.txt"... [PASSED]
( 95/125) Checking "094.txt"... [PASSED]
( 96/125) Checking "095.txt"... [PASSED]
( 97/125) Checking "096.txt"... [PASSED]
( 98/125) Checking "097.txt"... [PASSED]
( 99/125) Checking "098.txt"... [PASSED]
(100/125) Checking "099.txt"... [PASSED]
(101/125) Checking "100.txt"... [PASSED]
(102/125) Checking "101.txt"... [PASSED]
(103/125) Checking "102.txt"... [PASSED]
(104/125) Checking "103.txt"... [PASSED]
(105/125) Checking "104.txt"... [PASSED]
(106/125) Checking "105.txt"... [PASSED]
(107/125) Checking "106.txt"... [PASSED]
(108/125) Checking "107.txt"... [PASSED]
(109/125) Checking "108.txt"... [PASSED]
(110/125) Checking "109.txt"... [PASSED]
(111/125) Checking "110.txt"... [PASSED]
(112/125) Checking "111.txt"... [PASSED]
(113/125) Checking "112.txt"... [PASSED]
(114/125) Checking "113.txt"... [PASSED]
(115/125) Checking "114.txt"... [PASSED]
(116/125) Checking "115.txt"... [PASSED]
(117/125) Checking "116.txt"... [PASSED]
(118/125) Checking "117.txt"... [PASSED]
(119/125) Checking "118.txt"... [PASSED]
(120/125) Checking "119.txt"... [PASSED]
(121/125) Checking "120.txt"... [PASSED]
(122/125) Checking "121.txt"... [PASSED]
(123/125) Checking "122.txt"... [PASSED]
(124/125) Checking "123.txt"... [PASSED]
(125/125) Checking "124.txt"... [PASSED]
125 out of 125 correct.
```

```
5:11:43 x 125 cnguyen:hydra0 hw3_dev ~/wrkspc/cool/cs461_mj/pa3
$
hydra0.eecs.utk.edu 0:zsh* 1:vim- 2:vim
```