# CMSC 430 - Project 3

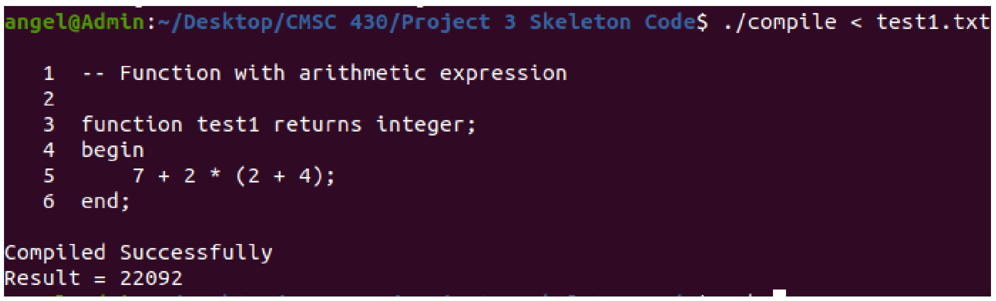
Modified Interpreter

Angel Lee

Nov 5, 2021

## How I approached Project 3

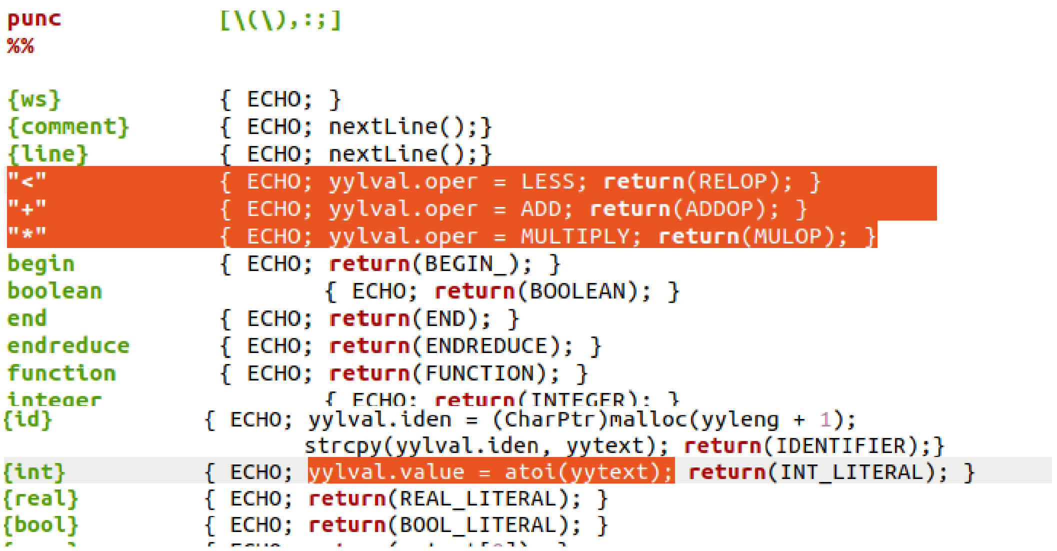
I combined code from Project 2 and transferred them over to Project 3, but had to modify some bison arguments to accommodate so the program can accept values for inputs. Some of the calculations were off when I ran test cases with the provided samples from the lectures, so I edited the code and compiled them line-by-line to see where issues were:



```
angel@Admin:~/Desktop/CMSC 430/Project 3 Skeleton Code$ ./compile < test1.txt

   1  -- Function with arithmetic expression
   2
   3  function test1 returns integer;
   4  begin
   5      7 + 2 * (2 + 4);
   6  end;

Compiled Successfully
Result = 22092
```

Results with wrong calculations

Scanners.l, values.h, and values.cc work together, so I edited the portions in scanner.l so the operations recall from the methods in values.h. I added more enum operators to values.h so they can call to values.cc:



```
punc            [\(\),:;]
%%

{ws}            { ECHO; }
{comment}       { ECHO; nextLine();}
{line}          { ECHO; nextLine();}
"<"             { ECHO; yylval.oper = LESS; return(RELOP); }
"+"             { ECHO; yylval.oper = ADD; return(ADDOP); }
"*"             { ECHO; yylval.oper = MULTIPLY; return(MULOP); }
begin           { ECHO; return(BEGIN_); }
boolean                 { ECHO; return(BOOLEAN); }
end             { ECHO; return(END); }
endreduce       { ECHO; return(ENDREDUCE); }
function        { ECHO; return(FUNCTION); }
integer                 { ECHO; return(INTEGER); }
{id}            { ECHO; yylval.iden = (CharPtr)malloc(yyleng + 1);
                        strcpy(yylval.iden, yytext); return(IDENTIFIER);}
{int}           { ECHO; yylval.value = atoi(yytext); return(INT_LITERAL); }
{real}          { ECHO; return(REAL_LITERAL); }
{bool}          { ECHO; return(BOOL_LITERAL); }
```

Arguments on scanner.l had to be updated

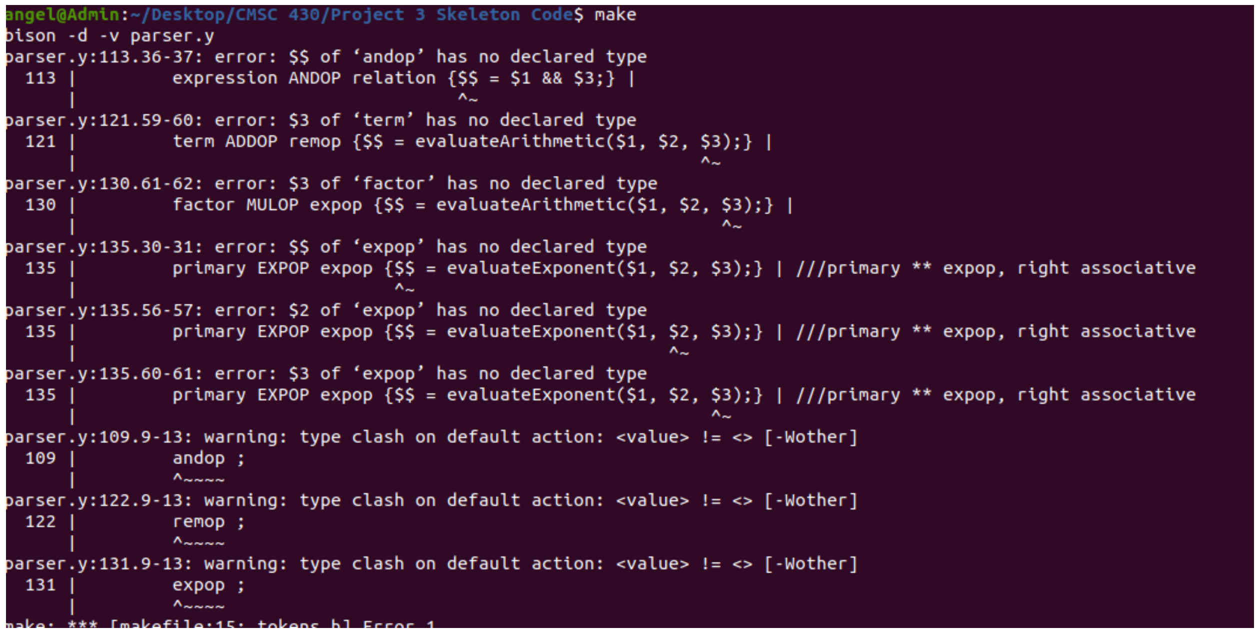
### Methods in values.cc

I added more methods and switch cases to values.cc, including: exponent method, subtract and divide switch cases, more relational operator switch cases, and a method to execute the exponent operation. I added the corresponding additions to values.h and scanner.l as well, so parser.y can use them.

I edited the methods in parser.y to include calls to values.h, and to include calls to the arithmetic methods in values.cc and values.h.

## Token values

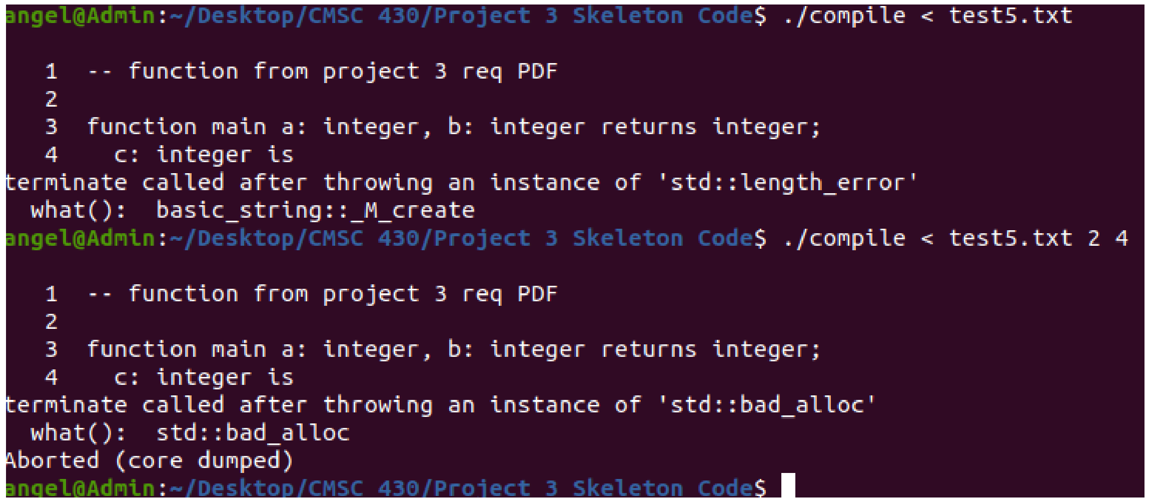
While inputting portions of Project 2 code into Project 3's parser in order to accomodate for the updates needed such as the addition of and/or operators, I was hit with the “has no declared type” error, and was trying to figure it out until I realized I need to put andop, remop, and expop in the %token(value) collection in parser.y:



```
angel@Admin:~/Desktop/CMSC 430/Project 3 Skeleton Code$ make
bison -d -v parser.y
parser.y:113.36-37: error: $$ of ‘andop’ has no declared type
  113 |         expression ANDOP relation {$$ = $1 && $3;} |
      |                                    ^~
parser.y:121.59-60: error: $3 of ‘term’ has no declared type
  121 |         term ADDOP remop {$$ = evaluateArithmetic($1, $2, $3);} |
      |                                                           ^~
parser.y:130.61-62: error: $3 of ‘factor’ has no declared type
  130 |         factor MULOP expop {$$ = evaluateArithmetic($1, $2, $3);} |
      |                                                             ^~
parser.y:135.30-31: error: $$ of ‘expop’ has no declared type
  135 |         primary EXPOP expop {$$ = evaluateExponent($1, $2, $3);} | ///primary ** expop, right associative
      |                              ^~
parser.y:135.56-57: error: $2 of ‘expop’ has no declared type
  135 |         primary EXPOP expop {$$ = evaluateExponent($1, $2, $3);} | ///primary ** expop, right associative
      |                                                        ^~
parser.y:135.60-61: error: $3 of ‘expop’ has no declared type
  135 |         primary EXPOP expop {$$ = evaluateExponent($1, $2, $3);} | ///primary ** expop, right associative
      |                                                            ^~
parser.y:109.9-13: warning: type clash on default action: <value> != <> [-Wother]
  109 |         andop ;
      |         ^~~~~
parser.y:122.9-13: warning: type clash on default action: <value> != <> [-Wother]
  122 |         remop ;
      |         ^~~~~
parser.y:131.9-13: warning: type clash on default action: <value> != <> [-Wother]
  131 |         expop ;
      |         ^~~~~
make: *** [makefile:15: tokens.h] Error 1
```

## Test sample from the Project 3 requirement PDF

I made a test file based off the test sample in the Project 3 requirement PDF and ran it to see what issues come up first before working on anything else:



```
angel@Admin:~/Desktop/CMSC 430/Project 3 Skeleton Code$ ./compile < test5.txt

   1  -- function from project 3 req PDF
   2
   3  function main a: integer, b: integer returns integer;
   4    c: integer is
terminate called after throwing an instance of 'std::length_error'
  what():  basic_string::_M_create
angel@Admin:~/Desktop/CMSC 430/Project 3 Skeleton Code$ ./compile < test5.txt 2 4

   1  -- function from project 3 req PDF
   2
   3  function main a: integer, b: integer returns integer;
   4    c: integer is
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted (core dumped)
angel@Admin:~/Desktop/CMSC 430/Project 3 Skeleton Code$
```

I noticed that it stops when the "if/else" statements come up, so I knew that the parser hasn't yet learned how to parse those, so I needed to transfer that grammar rule from Project 2 (since I already worked on it) and take it over to Project 3's skeleton code and then give it actions in bison to perform.

**If else method**

While looking at the 2nd option (reduce…) of "statement" and looking at the "reductions" method, I noticed that the "case _ is when.." option needed the ability to expand as well. I took at look at how the reductions method was written in *values.h* file and *values.cc* file, and then started new methods in the two latter files for the "if then else" and "case" grammar rules. Making the "if else" method was pretty simple because it pretty much means "if true/false then ____ else _____" and I made a method where the method relies on booleans: if true, then ___, if false, else____.

**Unix/Compiler input**:

I tried tackling the case parsing methods but they were more complicated so I left that aside for later. In order to even work on the case method, I knew I had to figure out how to take the integer input from the unix compiler. At the 'variable' grammar rule section in the parser, there was *{symbols.insert ($1, $5);}* which I saw as a clue on what I should do.

I assumed that I should be able to take the argument vectors from the compiler as inputs and then put them into initialized integer variable array or a queue and then be able to take the values from the array/queue and then put that into the symbols table.

I first initialized variables for the argument vector integers in the %{ }% area, so the main method can take the argument vector, add the value to the arg variable, and the arg variable can be used in the parameter method to be input into symbols.h. I first tried with the "if argument count  >= 2" but then the code got really long and inefficient, and it wasn't working.

I decided to use queue, and initialized an integer variable "arg" to take the input from the queue.front(), but spent too much time trying to figure out why it wasn't working. When I removed the use of the integer variable and directly put queue.front() as the input into the parameters' symbol insert like *{symbols.insert ($1, argQueue.front());}*, and used the pop function to remove the value from queue right after, then it worked. It is such a simple elegant code to just use queue.

**Case statements**

Since reductions can have multiple statements and there is no end to how many statements it can have, and it either multiplies or adds those statements, I assumed that case probably needed the same thing - the ability to expand. Reduction grammar was written as "reductions: reductions statement_" and the reductions can keep expanding into multiple statements (reductions statement_ statement_ statement_…." While using the operator that was written before it in statement's second option (REDUCE operator reductions ENDREDUCE).

Hence in statement’s 4th option (CASE expression IS case_ OTHERS ARROW statement_ ENDCASE) I gave case_ an option to continue expanding upon itself like *case_ case case case…*

And for the 'case' grammar rule, I knew I had to make a case method in *values.cc* that would match the input and response accordingly, like if case doesn’t match input, skip or return a null value, until case matches input then return the statement to be the value for case_.

**Case statement parsing**

To do the case statement parsing, there were several steps I needed to take. This required multiple actions happening on one grammar rule (CASE expression…) and not only that, I also needed a way to look at the returned values for each integer case. I first thought I needed to figure out a way to first take the expression/identifier and use it as a reference to find its value in the symbol table and compare it to the cases, only to realize later that it was unnecessary and if an identifier is called, it will already give an integer result for the methods in values.cc to work with. I spent more than 10 hours figuring this out until I realized it was unnecessary.

In order to let the program pick and choose between the cases and the Other option, I needed a method in values.cc that would look at the case method in the parser, and named it evaluateCase. Then, I needed a couple more for the case grammar rule under the statement method. I needed multiple actions inside the {C statements} for the case grammar rule, and I learned about mid-action rules, which is what I needed (see #3 in reference at bottom of this document).

One action/method for the 'case' grammar rule would take the value of the identifier in the case grammar rule (named *caseIdentifier()* in *values.cc*), then caseIdentifier would extract the value from the identifier token make it available on the values.cc’s end so the case-parsing method on values.cc can then use that value to compare it to the cases’ int_literals (case to reference that will look at the “when _ arrow ___” and return the statement value.

I switched caseIdentifier from int to void, because it wasn’t returning any new results but instead just passing value to values.cc for its use by other methods.

The *evaluateCase()* method in *values.cc* works with the 'case' grammar rule in the parser. It compares the literal to the provided identifier’s value, and if they match, then it returns the statement/value as the result. If they don’t match, it returns 0.

I made the *evaluateCases()* method for the 'case_' grammar rule of the parser, and what this does is combine the sum of all the case option results (except Others). This is fine because if a case’s integer literal didn’t match the value of the identifier/expression, then the result would be 0 anyway, so all the 0’s of the unmatched cases plus the result of the matched case would still equal the matched cases’ statement result.

Then at *evaluateCaseStatement(),* the method then picks between the statement result of all the cases from evaluateCases, versus the Other statement’s result. If one of the cases matched with the found value back in evaluateCase(), then the evaluateCaseStatement would pick cases's result, otherwise it would pick the Other's statement.

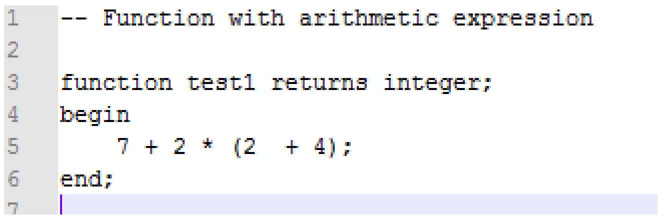
## Test Plan:

I made 7 test files (included in the ZIP file): 4 from the video lecture, 5th from the Project Requirement PDF, and 6/7th are variations of the PDF test case to specifically test certain parts of the program. Test 5 actually has multiple argument vector inputs however to see if it actually worked. To see if my modification of the skeletal codes worked, I compared my modified code to the skeletal code.

| **Test case** | **Expected output** | **Met expectation?** |
|---|---|---|
| 1, from video lecture | Compiled Successfully, result 19 | Yes |
| 2, from video lecture | Compiled Successfully, result 1 | Yes |
| 3, from video lecture | Compiled Successfully, result 17 | Yes |
| 4, from video lecture | Compiled Successfully, result 65 | Yes |
| 5, from Project req. PDF with argument inputs "2 4", "1 4", and "3 4" | Compiled Successfully with the result 0, 1, and 4 respectively to the inputs | Yes |
| 6 from Project req. PDF but edited to test if/else method | Compiled Successfully, result 26 | Yes |
| 7, from Project req. PDF but edited to test argument input | Compiled Successfully, result 9 | Yes |

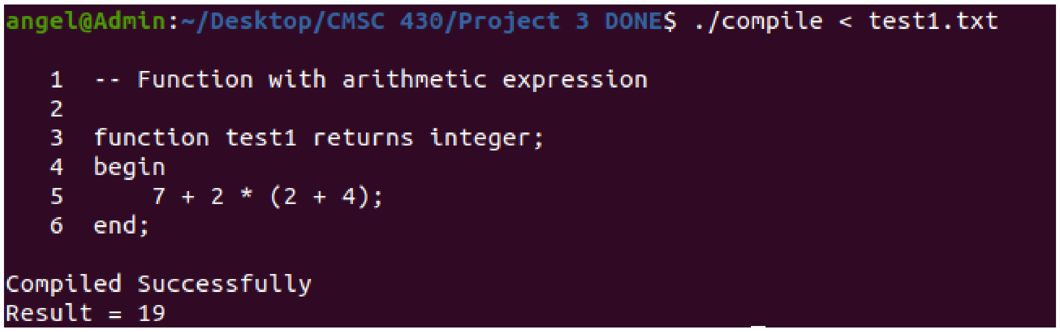
**Test1:**

This tests if the parser reads and passes valid arithmetic expressions and operators.



```
-- Function with arithmetic expression

function test1 returns integer;
begin
    7 + 2 * (2  + 4);
end;
```

Compilation for test 1:

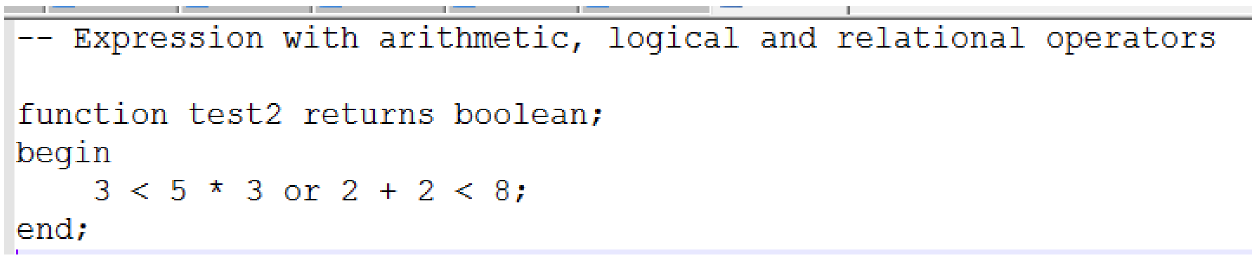


```
angel@Admin:~/Desktop/CMSC 430/Project 3 DONE$ ./compile < test1.txt

   1  -- Function with arithmetic expression
   2
   3  function test1 returns integer;
   4  begin
   5      7 + 2 * (2 + 4);
   6  end;

Compiled Successfully
Result = 19
```

This compiled successfully as there were no issues due to no syntax errors, and the result calculated correctly as 19 since 7 + 2*(2+4) = 7+2*6 = 7+12 = 19

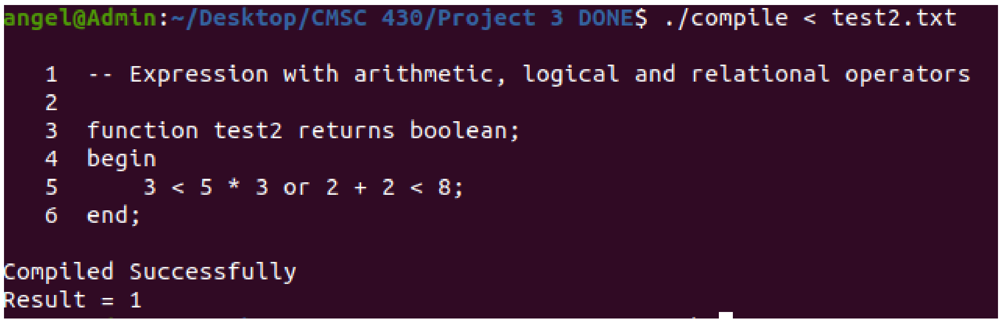
**Test 2:**

This tests arithmetic, logical, and relational operators:



```
-- Expression with arithmetic, logical and relational operators

function test2 returns boolean;
begin
    3 < 5 * 3 or 2 + 2 < 8;
end;
```

Compilation for test 2:

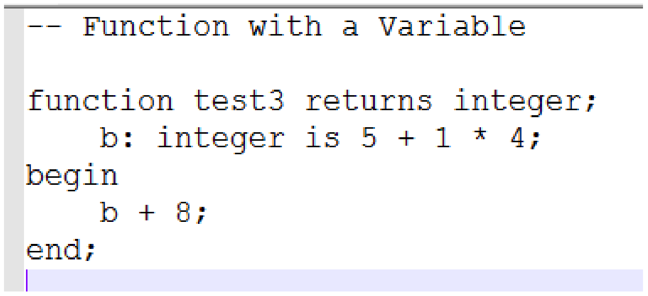


```
angel@Admin:~/Desktop/CMSC 430/Project 3 DONE$ ./compile < test2.txt

   1  -- Expression with arithmetic, logical and relational operators
   2
   3  function test2 returns boolean;
   4  begin
   5      3 < 5 * 3 or 2 + 2 < 8;
   6  end;

Compiled Successfully
Result = 1
```

This compiled successfully as there were no issues due to no syntax errors, and the operator precendence performed as it should since 5*3 took first priority in 3<5*3, and then 2+2 took priority in 2+2<8 before being calculated and giving out boolean results to then be compared by "or". *(3<15) or (4<8)* becomes *1 or 1*, which is 1.

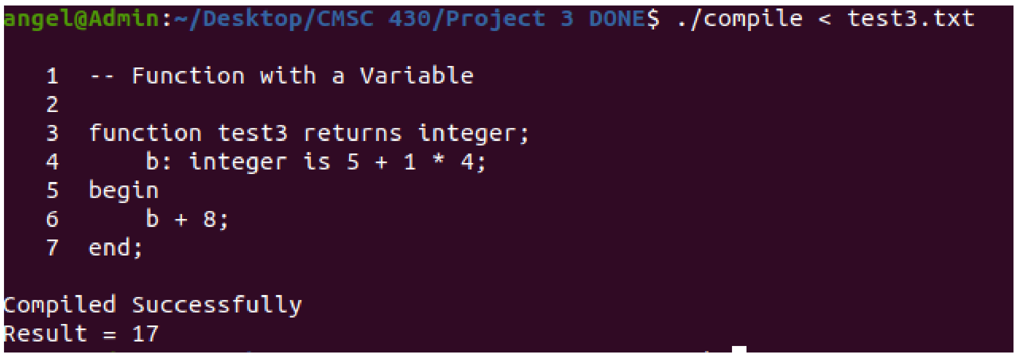
**Test3:**

This tests whether variables have taken value to be used in calculation:



```
-- Function with a Variable

function test3 returns integer;
    b: integer is 5 + 1 * 4;
begin
    b + 8;
end;
```

Compilation for test 3:

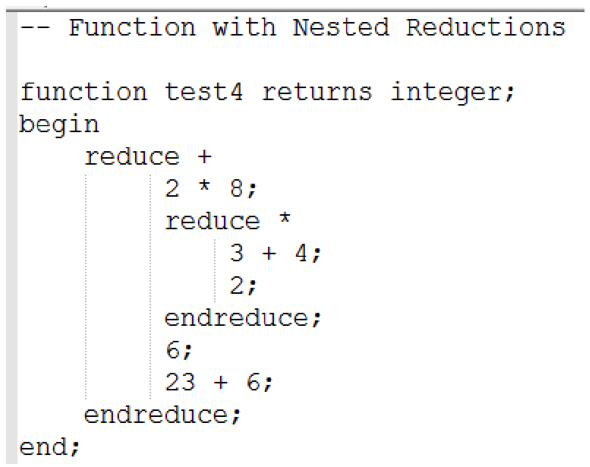


```
angel@Admin:~/Desktop/CMSC 430/Project 3 DONE$ ./compile < test3.txt

   1  -- Function with a Variable
   2
   3  function test3 returns integer;
   4      b: integer is 5 + 1 * 4;
   5  begin
   6      b + 8;
   7  end;

Compiled Successfully
Result = 17
```

This compiled successfully as there were no issues due to no syntax errors, and the variable "b" has taken value (5+1*4 = 9) as it should. b = 9 and 9+8 = 17, so it calculated correctly.

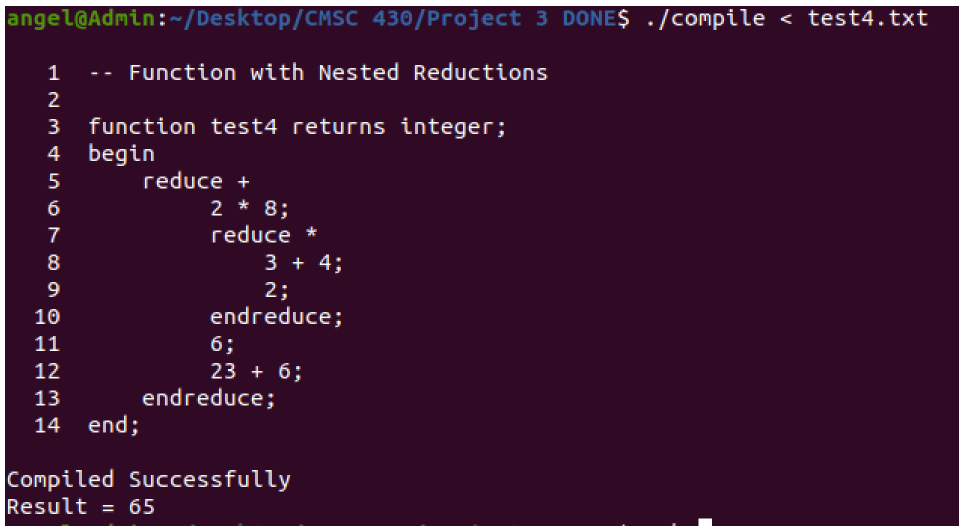
**Test 4:**

This tests the reduction method, and since all the syntax is correct with no lexical issues, it should compile.



```
-- Function with Nested Reductions

function test4 returns integer;
begin
    reduce +
         2 * 8;
         reduce *
             3 + 4;
             2;
         endreduce;
         6;
         23 + 6;
    endreduce;
end;
```

Compilation:

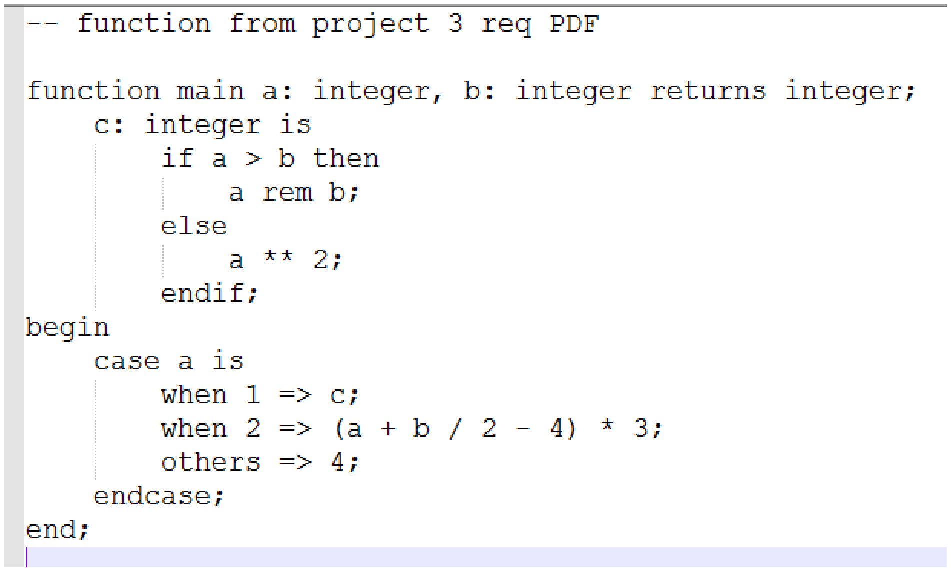


```
angel@Admin:~/Desktop/CMSC 430/Project 3 DONE$ ./compile < test4.txt

   -- Function with Nested Reductions

   function test4 returns integer;
   begin
       reduce +
            2 * 8;
            reduce *
                3 + 4;
                2;
            endreduce;
            6;
            23 + 6;
       endreduce;
   end;

Compiled Successfully
Result = 65
```

This compiled successfully as there were no issues due to no syntax errors. The first reduction is (3+4)*2 = 14, and the second reduction is (2*8) + 14 + 6 + (23+6) which is 65. It was calculated correctly.

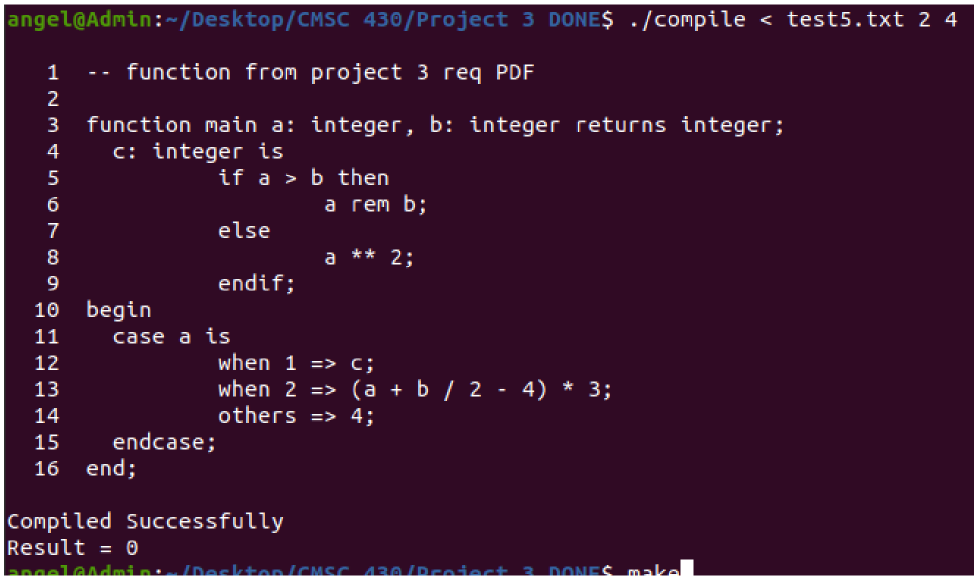
**Test5:**

This is the test sample from the Project 3 requirement PDF:



```
-- function from project 3 req PDF

function main a: integer, b: integer returns integer;
    c: integer is
        if a > b then
            a rem b;
        else
            a ** 2;
        endif;
begin
    case a is
        when 1 => c;
        when 2 => (a + b / 2 - 4) * 3;
        others => 4;
    endcase;
end;
```

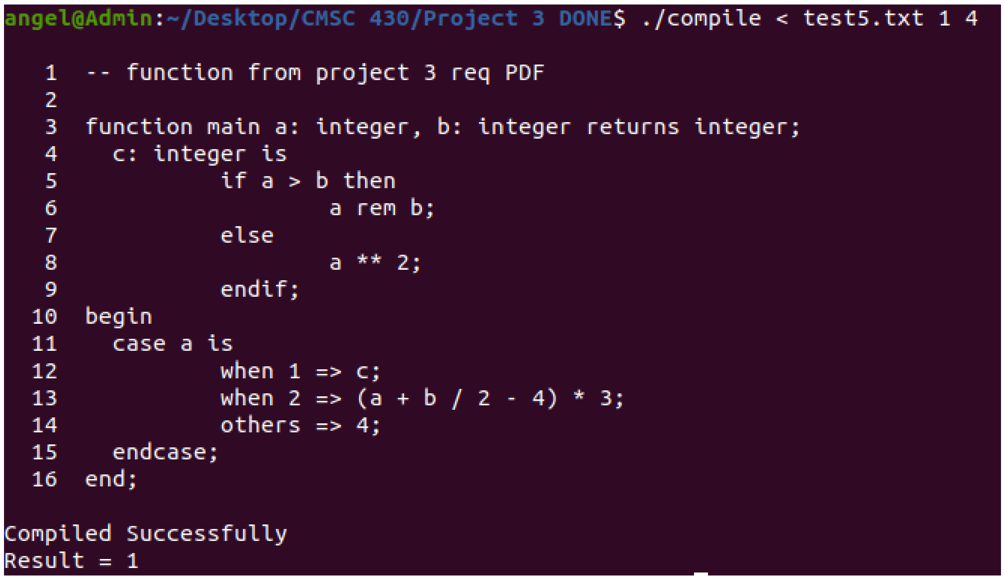
There are 3 compilations, as I put 3 different set of arguments in the compiler. First one was the "2 4" as shown in the PDF, and it compiled exactly as it looks:



```
angel@Admin:~/Desktop/CMSC 430/Project 3 DONE$ ./compile < test5.txt 2 4

  -- function from project 3 req PDF

  function main a: integer, b: integer returns integer;
    c: integer is
            if a > b then
                    a rem b;
            else
                    a ** 2;
            endif;
  begin
    case a is
            when 1 => c;
            when 2 => (a + b / 2 - 4) * 3;
            others => 4;
    endcase;
  end;

Compiled Successfully
Result = 0
angel@Admin:~/Desktop/CMSC 430/Project 3 DONE$ make
```

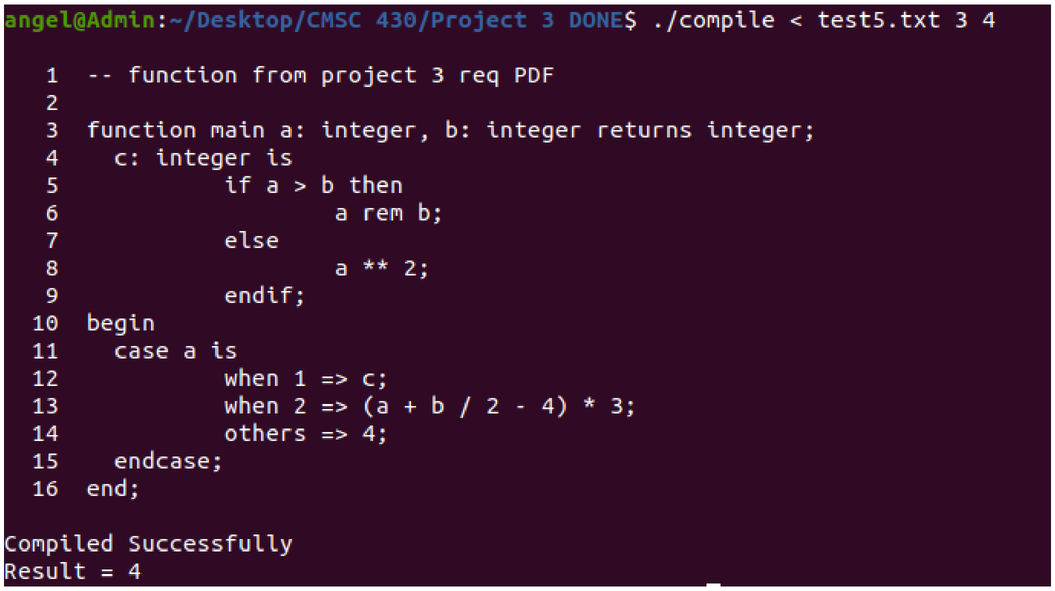
Result was 0, because a = 2, and when it's 2, the result is (2 + 2/2 - 4) * 3 = 0.

Second compilation had arguments "1 4":


```
angel@Admin:~/Desktop/CMSC 430/Project 3 DONE$ ./compile < test5.txt 1 4

  -- function from project 3 req PDF

  function main a: integer, b: integer returns integer;
    c: integer is
            if a > b then
                    a rem b;
            else
                    a ** 2;
            endif;
  begin
    case a is
            when 1 => c;
            when 2 => (a + b / 2 - 4) * 3;
            others => 4;
    endcase;
  end;

Compiled Successfully
Result = 1
```

Result was 1, since a = 1, and when 1, it is c. And c = a^2 = 1^2 = 1 since a is not bigger than b.

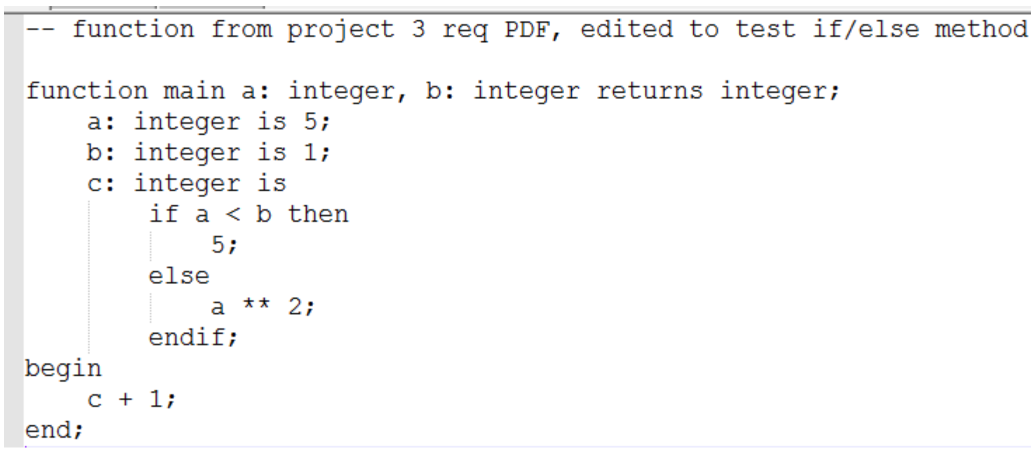
Third compilation had arguments "3 4":


```
angel@Admin:~/Desktop/CMSC 430/Project 3 DONE$ ./compile < test5.txt 3 4

  -- function from project 3 req PDF

  function main a: integer, b: integer returns integer;
    c: integer is
            if a > b then
                    a rem b;
            else
                    a ** 2;
            endif;
  begin
    case a is
            when 1 => c;
            when 2 => (a + b / 2 - 4) * 3;
            others => 4;
    endcase;
  end;

Compiled Successfully
Result = 4
```

Result is 4, because a = 3 and since it's not 1 or 2, the option is "others" which result in 4.

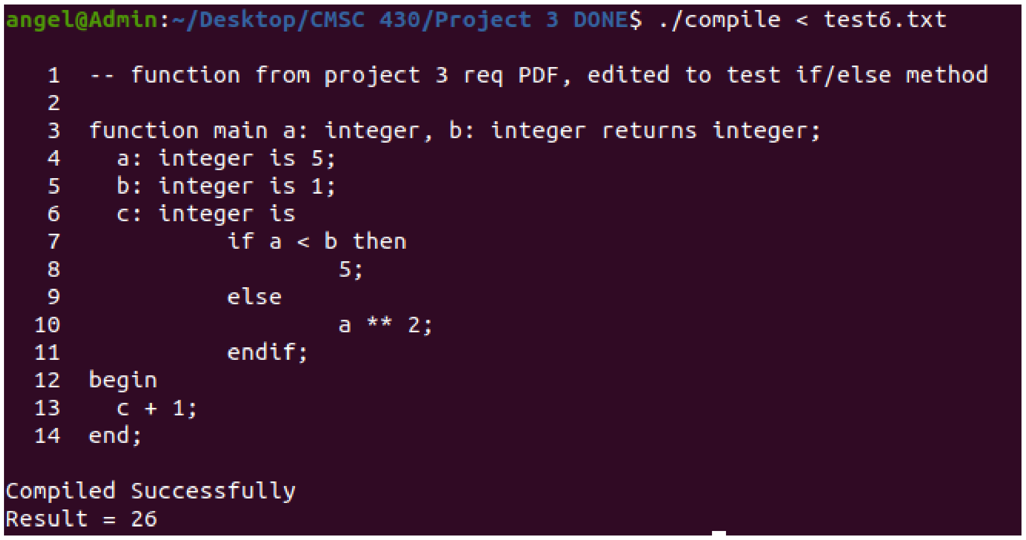
**Test6:**

The test case is an edited sample from the PDF. I edited it to test the if/else method while I was still working on the if/else method.



```
-- function from project 3 req PDF, edited to test if/else method

function main a: integer, b: integer returns integer;
    a: integer is 5;
    b: integer is 1;
    c: integer is
        if a < b then
            5;
        else
            a ** 2;
        endif;
begin
    c + 1;
end;
```

Compilation:

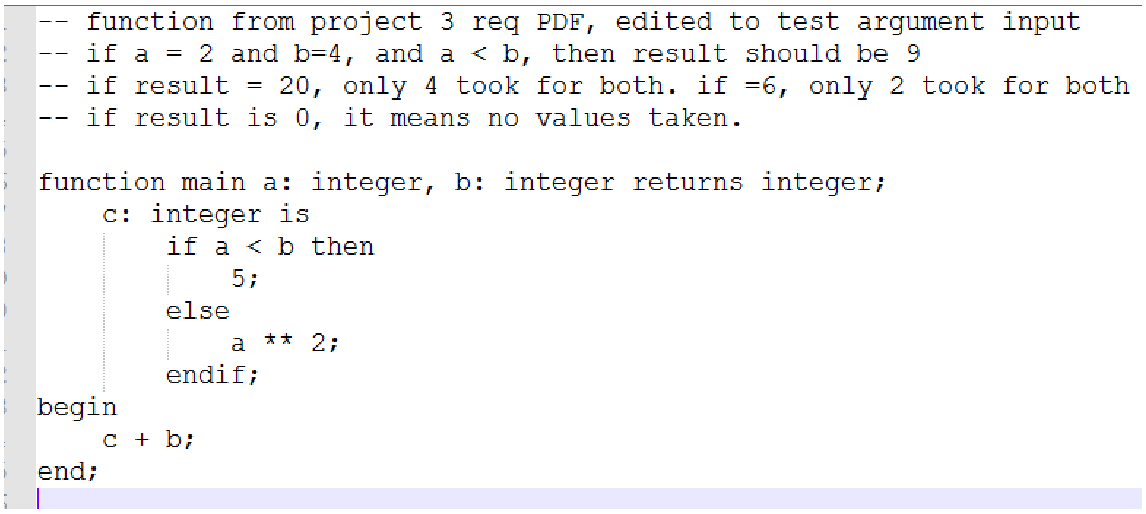


```
angel@Admin:~/Desktop/CMSC 430/Project 3 DONE$ ./compile < test6.txt

   1  -- function from project 3 req PDF, edited to test if/else method
   2
   3  function main a: integer, b: integer returns integer;
   4    a: integer is 5;
   5    b: integer is 1;
   6    c: integer is
   7            if a < b then
   8                    5;
   9            else
  10                    a ** 2;
  11            endif;
  12  begin
  13    c + 1;
  14  end;

Compiled Successfully
Result = 26
```

Result is 26, because even when there isn't an argument input from the compiler line (I made sure it didn't give any errors if there WASN'T argument inputs), the variable values are provided in the variable lines. Since a=5 is not smaller than b = 1, c = a^2 = 5^2 = 25. Then c + 1 = 25 + 1 = 26. It calculated correctly.

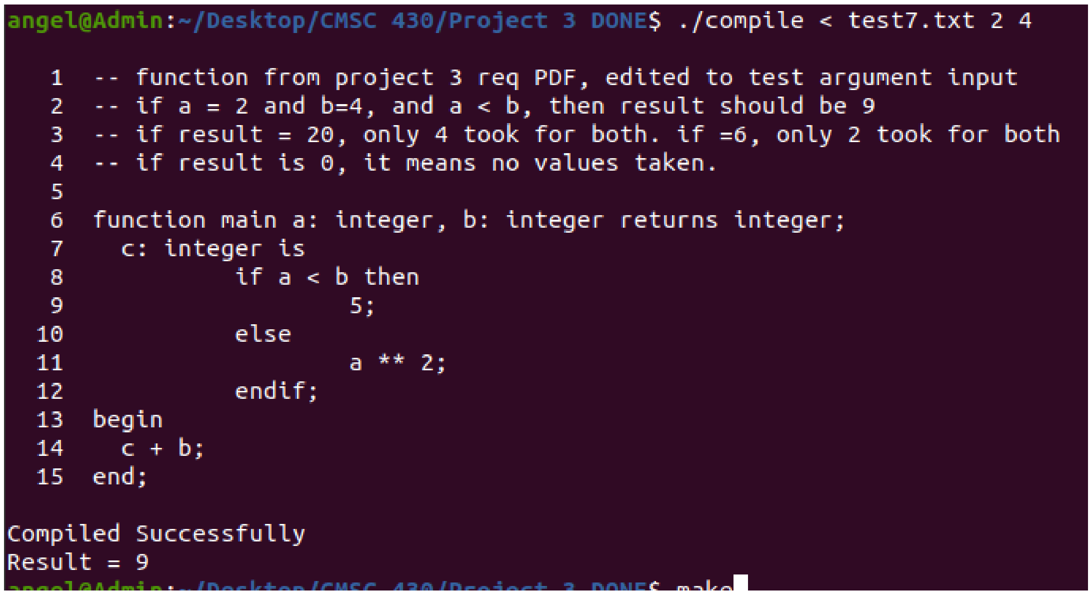
**Test7:**

Based off the test case from the PDF as well. I used this while testing whether the argument inputs from the compiler line was taken into the argument queue for the parameters as expected. Sometimes there were issues where either the first or last arguments were put in queue but not the other.



```
-- function from project 3 req PDF, edited to test argument input
-- if a = 2 and b=4, and a < b, then result should be 9
-- if result = 20, only 4 took for both. if =6, only 2 took for both
-- if result is 0, it means no values taken.

function main a: integer, b: integer returns integer;
    c: integer is
        if a < b then
            5;
        else
            a ** 2;
        endif;
begin
    c + b;
end;
```

Compilation:



```
angel@Admin:~/Desktop/CMSC 430/Project 3 DONE$ ./compile < test7.txt 2 4

   1  -- function from project 3 req PDF, edited to test argument input
   2  -- if a = 2 and b=4, and a < b, then result should be 9
   3  -- if result = 20, only 4 took for both. if =6, only 2 took for both
   4  -- if result is 0, it means no values taken.
   5  
   6  function main a: integer, b: integer returns integer;
   7    c: integer is
   8            if a < b then
   9                    5;
  10            else
  11                    a ** 2;
  12            endif;
  13  begin
  14    c + b;
  15  end;

Compiled Successfully
Result = 9
angel@Admin:~/Desktop/CMSC 430/Project 3 DONE$ make
```

The input were taken in as expected, and since a < b, c = 5. Then c + b = 5 + 4 = 9. Everything calculated correctly.

**Lessons Learned and any improvements that could be made**

A lot of my lesson learned is in the "how I approached the project" portion above. I would say coding-wise, my biggest lesson here is using the mid-action rules (see #3 in reference) in bison and even using multiple actions on one grammar rule. I don't know if this was explained in the yacc tutorial PDF provided in-class, but I didn't find much info about it so I had to go online to see if there were ways to extract information FROM the grammar rule while parsing it.

Another lesson I learned is how precise and specific one must be while programming and parsing, but this is something I already knew. This project just put this in my face more. I spent hours trying to make the queue work for the argument vector inputs, and can't figure out why it wasn't taking - all it took was not using the integer variable that I first made to put the arguments in. Another thing that took forever and didn't need to was the case statement parsing, where I forgot that if an identifier/expression (such as "a") is called, it will automatically return the corresponding value (such as 2 if a = 2 in the symbol table) and I spent more than 10 hours trying to figure out how to read the symbol table instead.

**References I used:**

1. http://dinosaur.compilertools.net/bison/bison_6.html
2. http://dinosaur.compilertools.net/bison/bison_6.html#SEC46
3. http://dinosaur.compilertools.net/bison/bison_6.html#SEC48
4. https://www.gnu.org/software/bison/manual/html_node/Actions.html
5. https://www.gnu.org/software/bison/manual/bison.html