COSC 461/561
cexpr - calculator for C integer expressions

Your assignment is to write a recursive descent parser that implements a calculator for C integer expressions. The *cexpr* calculator will employ the *cscan* scanner from your first assignment to scan tokens in the input, including integer numbers and identifiers. Note that some of the tokens identified by *cscan* will not be used in this assignment.

The calculator should process statements containing integer expressions until it encounters EOF or invalid syntax. Each statement is terminated by a semicolon. The identifiers in each statement will always correspond to integer variables. If a variable is used in a statement before it is assigned, it should be initialized with a value of 0.

The calculator should print the results of each statement to standard output. Specifically, if the statement assigns one or more integer variables, it should print the names of the variables and their assigned integer values. If the statement calculates an integer expression without making any assignment, it should only print the integer result of the full calculation. In statements that assign integer variables, no operators can precede an assignment operator except another assignment operator

The table below defines the C operators you need to implement. The operators are listed in order of decreasing precedence.

symbols	comments	type	assoc
( )	parentheses	unary	n/a
∼ −	bitwise not, negation	unary	right
∗ / %	mul, div, mod	binary	left
+ −	add, sub	binary	left
<< >>	shifts	binary	left
&	bitwise and	binary	left
∧	bitwise xor	binary	left
|	bitwise or	binary	left
= += -= *= /= %= <<= >>= &= ∧= |=	assignment	binary	right

You should structure your solution as a recursive descent parser corresponding to the grammar at the end of this statement.

To avoid duplicating code unnecessarily, your solution should abstract common tasks for the different operators into common routines. For example, the routine that applies all assignment operations should take the following arguments: 1) a token corresponding to the assignment operator, 2) an l-value corresponding to the left operand, and 3) an r-value corresponding to the right operand. You will also need to employ *precedence climbing* so that your solution enforces the associativity and precedence described in the table above.

Additionally, your solution should detect error conditions, including invalid shift operands and divide (and mod) by zero. While our solution detects integer overflow, you are not required to implement integer overflow checking. When there is an error condition, you should print an appropriate message to standard out indicating the first type of error encountered rather than the value after the calculation. You should not perform any further calculations or assignments after encountering an error.

The starter package includes a working scanner library with an interface provided in scan.h. The package also includes a (mostly unimplemented) cexpr.c and makefile to help get you started. For this assignment, you are only allowed to only modify the cexpr.c file. The starter package also includes sample inputs as well as a reference executable (`ref_cexpr`) that you can use to test alternative inputs.

For submission, you should upload your modified cexpr.c file to the Canvas course website by 11:59pm on the assignment due date. Partial credits will be given for incomplete efforts. However, a program that does not compile or run will get 0 points. Point breakdown is below:

- syntax parsing of each expression / command (20)
- variable assignment (20)
- basic expression calculation (20)
- precedence / associativity of each operator is correct (20)
- error checking (shift operands, divide-by-zero) (10)
- efficient / elegant design (e.g., no unnecessary computation or restrictions) (10)

Grammar for *cexpr*:

$$
\begin{array}{lcl}
\textit{prog} & \rightarrow & \textit{stmts} \\
 & & \\
\textit{stmts} & \rightarrow & \textit{stmt}\ \textit{stmts} \\
\textit{stmts} & | & \epsilon \\
 & & \\
\textit{stmt} & \rightarrow & \textit{expr}\ \textbf{;} \\
\textit{stmt} & \rightarrow & \textbf{;} \\
 & & \\
\textit{expr} & \rightarrow & \textit{term} = \textit{expr} \\
 & | & \textit{term}\ |\!= \textit{expr} \\
 & | & \textit{term} \wedge = \textit{expr} \\
 & | & \textit{term}\ \textbf{\&}\!= \textit{expr} \\
 & | & \textit{term} <\!<\!= \textit{expr} \\
 & | & \textit{term} >\!>\!= \textit{expr} \\
 & | & \textit{term} + = \textit{expr} \\
 & | & \textit{term} - = \textit{expr} \\
 & | & \textit{term} * = \textit{expr} \\
 & | & \textit{term}\ / = \textit{expr} \\
 & | & \textit{term}\ \textbf{\%}\!= \textit{expr} \\
 & | & \textit{term}\ |\ \textit{expr} \\
 & | & \textit{term} \wedge \textit{expr} \\
 & | & \textit{term}\ \textbf{\&}\ \textit{expr} \\
 & | & \textit{term} << \textit{expr} \\
 & | & \textit{term} >> \textit{expr} \\
 & | & \textit{term} + \textit{expr} \\
 & | & \textit{term} - \textit{expr} \\
 & | & \textit{term} * \textit{expr} \\
 & | & \textit{term}\ /\ \textit{expr} \\
 & | & \textit{term}\ \textbf{\%}\ \textit{expr} \\
 & | & \textit{term} \\
 & & \\
\textit{term} & \rightarrow & (\ \textit{expr}\ ) \\
\textit{term} & | & -\ \textit{expr} \\
\textit{term} & | & \sim \textit{expr} \\
\textit{term} & | & \textbf{identifier} \\
\textit{term} & | & \textbf{int\_literal}
\end{array}
$$

```
Example Input:
a = 55-3;
b = c = a-42;
a+b*c;
a = 2;
a = (b-(a+c)*-5);
c/d;
d/c;
^D
```

```
Example Output:
$> a := 52
$> c := 10
$> b := 10
$> 152
$> a := 2
$> a := 70
error: attempted to divide by 0
$> 0
```