How to use this program:

Compile the Source:

Compile either 'DCG.prolog' or 'DCG_no_parse_tree.prolog'. Both run the same way, the difference is noted in the extra credit section.

```
['DCG.prolog'].
['DCG_no_parse_tree.prolog'].
```

Expression Evaluation:

This program is a simple calculator that can evaluate expressions. To evaluate an expression, use either the predicate evaluate/1 or evaluate/2. The evaluate/1 predicate evaluates an expression and prints the result. The evaluate/2 predicate evaluates the expression in the first argument and instantiates the second argument to the result.

```
?- evaluate([5, '+', 5]).
Result is 10
Yes
?- evaluate([5, '+', 5], X).
X = 10
```

The expressions given as arguments to the evaluate predicate, must be in the form of a list atoms, such as [tan, '(', 3.14, '/', 4, ')'].
Valid operators are '+', '-', '*', '/', '(', ')' and '^'.
Valid functions are tan, sin and cos.

Using Variables:

Variables can be assigned values using the assign/2 predicate. The first argument is the name of the variable that is being assigned and the second argument is the value of the variable.

```
?- assign(cat, 2).
cat = 2
Yes
?- evaluate([2, '^', 0.5], X), assign(root2, X).
root2 = 1.41421
X = 1.41421
```

The assign predicate can also take an expression as it's second argument.

```
?- assign(dog, [5, '^', 0.5]).
dog = 2.23607
Yes
```

To use variables in expressions, simply make the variable name as an element in the list.

```
?- evaluate([dog, '+', cat]).
Result is 4.23607
Yes
```

Context Free Grammar:

Example 2.7 from Scott's book:

expr → term | expr add_op term
term → factor | term mult_op factor
factor → id | number | - factor | (expr)

add_op → + | -
mult_op → * | /

The example in section 2.7 of Scott's book has left recursion. If such a grammar were implemented into a program, the program would form an infinite loop while trying to form an expr from another expr followed by an add_op and a term. The general solution to this problem is:

expr → term exprtail
exprtail → ε
exprtail → add_op term exprtail

My version of Scott's grammar, including rules for exponents and functions:

expr → term exprtail | func factor
exprtail → ε | add_op term exprtail
term → power termtail
termtail → ε | mult_op power termtail
power → factor powertail
powertail → ε | pow_op factor powertail
factor → id | number | add_op factor | (expr)
add_op → + | -
mult_op → * | /
pow_op → ^
func → tan | sin | cos

Extra Credit:

Exponents: Support for exponents has been added with correct order of operations:

$$2^{3^4} = 2^{(3^4)} \neq (2^3)^4$$

Functions: Support for basic trigonometric functions has been added.

Variables: Support for the assignment and use of variables in expressions.

Evaluation without Parse Tree: The 'DCG_no_parse_tree.prolog' evaluates expressions without first creating a parse tree. This evaluates the expression with fewer inferences.