Compile and execute arithmetic expressions.

1 Program

You should write a driver program that takes a text file as a command line argument and then processes it using your compiler. The compiler should parse the input for a full arithmetic expression and generate a code tree representing that expression. The code tree is then used to generate machine code instructions to be executed later.

2 Basic Compiler Outline

The pseudo code of your compiler should be roughly as follows:

```
initialize the various modules of your code (lexical analyer, etc.)

while there's still input
   get/parse a code tree.  Print any errors generated in this process.

   print the code tree // Ultimately for debugging only

   // Code tree optimization goes here
   // but none needed for this assignment

   generate machine code instructions (the program) from the code tree

   execute the program

   delete the code tree

   cleanup the program

cleanup the compiler
```

3 Operators

The arithmetic operators that should be implemented for this assignment are

- addition and subtraction, using the '`+`' and '`-`' tokens
- multiplication and division, using the '`*`' and '`/`' tokens
- division remainder using the '`mod`' keyword
- exponentiation using the '`^`' token. For the purposes of this assignment, this operator should always return the value of the left operand, `a ^ b` should return the value of `a`. This is obviously arithmetically wrong, but will be fixed later.
- unary plus (which does nothing) and unary minus (negation) using the '`+`' and '`-`' tokens. It should not be possible to make a sequence of these operators, i.e. `+++-+---+5` is illegal.
- arithmetic expression grouping using parentheses, '`(`' and '`)`'

The arithmetic operators form a precedence hierarchy. For operators that have the same precedence, there is also an associativity for those operators indicating the order of evaluation.

For example, in the expression `a - b - c` if '-' is left associative, then the result is equivalent to `(a - b) - c`. However if '-' is right associative, then the expression is equivalent to `a - (b - c)`.

The following table shows the expected operator hierarchy and associativity for each level. The highest precedence is at the top of the table.

Operators	Associativity
`()` (Grouping)	N/A
`+, -` (Unary)	N/A
`^`	right
`*, /, mod`	left
`+, -` (Addition and subtraction)	left

4 Code Tree Generation

This is where the input file is parsed. We will use a top-down approach to create a *recursive descent* parser and code tree generation.

Creating a context-free grammar for parsing arithmetic expressions can be a very useful guide in developing this stage. There should be a function for each left hand side of a grammar rule. Processing the right hand sides will consist of calling other functions (grammar rules) and consuming terminal tokens from the file. Calling other functions will result in code trees being passed back. These represent subexpressions which will be combined with information in the current function to create a larger tree.

When a grammar rule is matched, the subtrees from calling other grammar rules will be combined using an operation indicated by some terminal at the current level. Take, for example, a routine that processes addition (and probably subtraction) and an input expression of `a + b`. Another routine will be called to process what will eventually come back as a code tree representing 'a'. The current routine will recognize that the next token to be processed is a '+'. It will consume the token and call the routine for processing 'b' (which is probably the same routine called for processing 'a'). When that routine comes back, the current routine will create an addition node, placing the subtrees for 'a' and 'b' as children to the addition node. This new tree will then be used for further processing (returned to caller, or possibly used within a loop in the current routine).

It is important to have a *current token* that is visible to all routines in this code generation module. All functions can see what is the token that is to be matched. If a routine matches the token, then it is *consumed*. This simply means storing any value associated with the current token (rare), or ignoring it (common). The current token is then replaced with a new token provided by the lexical analyzer.

5 Printing

You must have a routine for printing out your code tree. This is invaluable for debugging and for verifying that your code tree has the proper structure. This is most easily generated by a recursive routine.

6 Cleanup

You should have a routine for deleting a code tree.

7 Machine Code Generation

The machine code for the expression is generated by doing a post-order traversal of the code tree. Code is generated for each of the subexpressions of a node and then additional code is added to process the node itself using the values of the subexpressions.

Only a limited subset of the Intel machine instructions is needed for the code generation part of this assignment. The most useful instructions are `RETn`, `PUSH`, `POP`, `MOV`, `ADD`, `SUB`, `NEG`, `IMUL`, `IDIV`, and `CDQ`. the compiler will be <

8 Execution Model

The easiest thing to do at this point for evaluating the arithmetic expressions is to implement a stack machine. The code for integer constants should push that value onto the system stack. Arithmetic operators should pop one or two values from the stack into registers, perform the desired operation, and then push the result back onto the stack. At the end of the code, the final value should be popped from the stack into the EAX register, and then the code should return.

9 Implementation Hints

It's not required, but your code organization will be greatly enhanced if you place all the code tree generation routines in source code file(s) separate from the file(s) for machine code generation. Use header files to properly communicate resources between files.

There are two basic ways to implement this assignment: 1) Get all of the code tree generation working before doing the machine code generation or 2) doing the code tree generation for a particular operator followed immediately by the machine code generation for that same operator. Both approaches are valid. There's not much to recommend one over the other. (But the author of this assignment did the first approach.)