You are to construct a compiler for the nano-C language. This will be done in a sequence of manageable steps. The final low-level code is to be executed by a virtual machine interpreter, which you are also to implement.

You may write the compiler in any programming language, but if you want to use the lex and yacc tools, they you should use C or C++. There are also lex and yacc like tools that generate Java or Standard ML code, so those languages are also reasonable choices.

You may work in groups of two. Larger groups are *not* permitted.

1 Assignment 1: Scanner (deadline: February 4)

Write a scanner for nano-C. Write three test programs which are to be used in all assignments. One of these is to be recursive and one is to have nested while statements. You may use lex to construct the scanner.

To validate the operation of the scanner, write a “fake” parser which repeatedly calls the scanner and prints the type and attribute (if any) of each token to the standard output. For example, for the statement x = 5; the fake parser might print:

```
IDENT x
ASSIGN
INTCONST 5
SEMI
```

2 Assignment 2: Parser (deadline: February 18)

Write a parser for nano-C. First you must augment the given nano-C grammar to properly express associativity and precedence for the arithmetic operators. Then you must convert the grammar to a form suitable for the parsing method you have chosen: you may use yacc to generate a LALR(1) parser, or you may write a recursive-descent parser by hand.

The parser shall construct an abstract syntax tree to represent the structure of the program. Therefore you must also define appropriate data types to represent abstract syntax trees for nano-C programs.
Assignment 3: Type checking and Intermediate code (deadline: March 4)

Implement type checking and translation to low-level intermediate code. The input is the abstract syntax tree produced by the parser. The output is a list of instructions for a simple virtual machine.

Type checking should traverse the abstract syntax tree, and verify that declarations and uses of identifiers follow the rules of nano-C.

The translation to intermediate code consists of a traversal over the abstract syntax tree, in which expressions and statements are converted to sequences of simple instructions. To implement recursion, a stack must be introduced for storing local variables and return addresses; this also affects the translation of calls, returns, and accesses to formal parameters and local variables.

Type checking and translation can be combined into a single module.

Assignment 4: Executing the code (deadline: March 25)

Implement an interpreter for the low-level intermediate code, and use it to execute your compiled nano-C programs. For simplicity, the interpreter module should be included in the compiler, and it should operate on the intermediate code data structure you generate.

General

Solutions to the assignments must contain the following:

- Title page with your name(s), the course name, and the number of the assignment.
- An overview on how you approached and solved the assignment.
- A listing of your code.

For each assignment, the rule is that errors in its input must be detected and reported: the scanner must report lexical errors, the parser must report syntax errors, and so on. You are permitted to halt your compiler after the first error.

You are permitted to ignore the issue of reclaiming unused memory in your compiler. That is, you may allocate memory dynamically without ever freeing it explicitly. (While not a recommended programming style, writing a compiler for the first time is complex enough.)