The following represents the EBNF grammar for a very simple language, called C--:

- **program**: {stmt}
- **stmt**: var_dec ; | assign ; | read_stat ; | write_stat ; | if_stmt ; | while_stmt ;
- **var_dec**: type var
- **assign**: var '=' expr
- **expr**: add_expr
- **add_expr**: mul_expr {('+'|'-') mul_expr}
- **mul_expr**: simple_expr {('*'/'/'%') simple_expr}
- **simple_expr**: id | var | ('(' expr ')')
- **read_stat**: "READ" '(' expr ')
- **write_stat**: "PRINT" '(' expr ')
- **type**: "int" | "float" | "boolean"
- **id**: intnumber | floatnumber
- **intnumber**: Digit | Digit intnumber
- **floatnumber**: Digit intnumber
- **Digit**: [0-9]+ | "0" | "1";
- **var**: [A-Z, a-z]+ | program [ block ]
- **if_stmt**: "if" bool_stmt ':' [block] [ "else:" [block] ] "end if"
- **while_stmt**: "while" bool_stmt "do" [block] "end while"
- **bool_stmt**: and_stmt | rel_stmt | boolean
- **and_stmt**: bool_stmt {("and"|"or") bool_stmt}
- **rel_stmt**: simple_expr ("<"|"<="|"=") simple_expr

**Please note the following about C--:**
- C-- variables are not case sensitive: Var and var refer to the same variable.
  - All variables are declared at the beginning of your program.
• **Sample input data:**

```c
{ 
    Int x;
    Int y;
    Int z;
    X = 10;
    Y = 20;
    print (x);
    print (x+y);
    read(z);
    z = z + x + y;
    print (z);
}

{ 
    if 5 > 1:
        
        if (4>3) and (2 <5):
            
                print 1;
                print 2;
            
        else:
            
                print 3;
                print (3+1)
        
        end if;
    
    end if;
}
```

This is a multi-phase project. In this first phase, we would like to develop a lexical analyzer for C—using C++. The other phases of the project will be given in the next assignment.

The **lexical analyzer** (sometimes referred as or recognizer or simply as tokenizer) reads a string of characters and checks if a valid token in the grammar.

**Lexical Analysis terminology:**
- **Token:**
  - Terminal symbol in a grammar
  - Classes of sequences of characters with a collective meaning, e.g., IDENT
  - Constants, Operators, Punctuation, Reserved words (keywords)
Lexeme:
  - character sequence matched by an instance of the token, e.g. "sqrt"

Given the following expression:

```
total = subtotal1 * 12;
```

- Lexical Analyzer step 1: group a stream of characters into lexemes:

```
t o t a l = s u b T o t a l 1 * 1 2 ;
```

- Lexical Analyzer step 2: Convert lexemes into tokens

```
t o t a l = s u b t o t a l 1 * 1 2 ;
```

The lexical analyzer should return the following:

<table>
<thead>
<tr>
<th>Token (Also known as Lexeme)</th>
<th>Token Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>VAR_CODE (identifier)</td>
</tr>
<tr>
<td>=</td>
<td>ASSIGN_OP</td>
</tr>
<tr>
<td>Subtotal1</td>
<td>VAR_CODE</td>
</tr>
<tr>
<td>*</td>
<td>MULTIPLY_OP</td>
</tr>
<tr>
<td>12</td>
<td>DIGIT_CODE (integer literal)</td>
</tr>
<tr>
<td>;</td>
<td>SEMICOLON</td>
</tr>
</tbody>
</table>

- Your lexical analyzer should include the following:
  - Extract tokens from a given input string and generate the type of the token.
  - Skip comments and blanks
  - You may need a lookup table that stores all the reserved words of your grammar. This will be used to check whether a token is a reserved word.
  - Insert user-defined tokens into a symbol table, which will be used by other components.
  - Detect syntactic errors in tokens and report such errors.
  - Your lexical analyzer will always put the code of the next token in a variable called `nextToken`. For example, if the token is an integer, `nextToken` will be equal to `DIGIT_CODE`.

- Develop your lexical analyzer is C++