Project 1: Writing a lexical analyzer for the BigAdd Language

The lexical analyzer for the BigAdd language works from the command line with the command LA, and takes the script file's name as the only attribute. The script file is assumed to have the extension *.ba* .

example: The command `c:\> la myscript`
must load the script file called myscript.ba and perform lexical analysis on it. The results of the lexical analysis should be written into a file with the same name and ".lx" extension. (For the example above The file myscript.lx will be created) This file should contain a suitable representation of a token at each line. (Each token should be written on a separate line)

example: Imagine myscript.ba contains the following code:

```
int size.
int sum.
move 5 to size.
loop size times     {ignore me, I am a comment}
[ out size, newline.
  add size to sum.
]
out newline, "Sum:", sum.
```

The content of myscript.lx has to be something like:

```
Keyword int
Identifier size
EndOfLine
Keyword int
Identifier sum
EndOfLine
Keyword move
IntConstant 5
Keyword to
Identifier size
EndOfLine
Keyword loop
Identifier size
Keyword times
OpenBlock
Keyword out
Identifier size
Seperator
Keyword newline
EndOfLine
Keyword add
Identifier size
Keyword to
Identifier sum
EndOfLine
CloseBlock
Keyword out
```

```
Keyword newline
Seperator
StringConstant "Sum:"
Seperator
Identifier sum
EndOfLine
```

If the lexical analyzer encounters an error, it should issue an error message on the screen. There are two error categories that such a lexical analyzer has to detect.
1- A big lexeme is just left open (a comment or a string constant that starts but does not terminate before the end of file)
2- An unrecognized character is detected in code

What to deliver: The teams have to deliver a report that contains their design and code. It should also inform about the tests that have been performed, together with some screenshots and output file samples. They should also deliver their source and execuable codes separately.

The BigAdd Language

BigAdd language is a small programming language that has been designed in order to add and subtract big integers programmatically and display the result on the screen. BigAdd is an interpreted language. The BigAdd interpreter works from the command line with the command BA, and takes the script file's name as the only attribute. The script file is assumed to have the extension *.ba* .

example: The command `c:\> ba myscript`
must load and execute the script file called myscript.ba

Lines of Code: Point ('.') is the end of line character. Lines can contain keywords, brackets, variables and constants. Any number of spaces or comments can be used between these elements. A program line can be divided between multiple text lines.
A line of code is one of the following:

- A variable declaration.
- An assignment statement.
- An addition statement.
- A subtraction statement.
- An output statement.
- A loop statement.

Comments:
Comments are written between curly braces { }
example: `{this is a comment}{ and this is a comment too}`

Data Types: The only data type for variables is the *integer*. Integer is a signed whole number that is represented as a **decimal data type**. An integer can be as big as **100 decimal digits**.

Strings exist only as constants and are used in the *out* statements only.

example: 123113, -5, 0, -314159 are valid integer representations.

3.14159 is not a valid integer (it is a real number)
3.0 is not a valid integer (decimal point should not be displayed)
3. is not a valid integer (decimal point should not be displayed)
- 5 is not a valid integer (there should be no blank between the minus sign and the first digit.
--5 is not a valid integer (only one minus sign allowed)
+5 is not a valid integer (plus sign is not allowed)

Variables: All variables should be declared as an integer. Variables must be declared before they are used. All variables are global (and static).
int <variable>.
Variable names are case sensitive and have a maximum length of 20 characters. Variable names start with a letter (alphabetical character) and continue with alphanumeric characters or an underscore character.
example: `int myVar.`

All variables are initialized with the value 0 at the time of creation.

Assignment Statement:
move <int_value> to <variable>.
example:
```
move 25 to myVar. {assigns 25 to myVar}
move myVar to yourVar. {assigns myVar to yourVar}
```

Addition statement:
add <int_value> to <variable>.
Increments the variable by int_value.

example: `add 2 to sum . {The value of sum increases by 2}`

Subtraction statement:
sub <int_value> from <variable>.

example: `sub t from total. {The value of total decreases by t}`

Output statement:
out <out_list>.
example: `out "The result is:",sum.`

An integer value is either a variable or a constant.
<int_value>→<variable>|<int_const>

An output list is a list of strings and integer values separated by commas.
<out_list>→<out_list>,<list_element>|<list_element>
<list_element>→<int_value>|<string>| newline

A string is any sequence of characters between two quotation marks.
example: "Hello, this is a string!"

Loop:
loop <int_value> times <line>
{OR}

loop <int_value> times <code_block>

A loop starts with the int_value and at each iteration decrements the value of int_value by one. If int_value is a variable, the value of the variable can be accessed and modified during the loop. After the loop the variable takes the value 0. The last iteration of the loop operates with the value 1.

example:

```
loop 10 times out "*". {writes 10 stars on the screen}
```

Code Block:

A code block is a list of code lines between square brackets. A code block can be used in a loop, interchangeably with a single line of code. By using code blocks, loops may be nested within each other.

example:

```
int size.
int sum.
move 5 to size.
loop size times
[ out size, newline.
  add size to sum.
]
out newline, "Sum:", sum.
```

Output is:

```
5
4
3
2
1
Sum:15
```

Errors: Error detection is an important aspect of a successful BigAdd implementation. To be able to get a high rating from an interpreter project, it should be able to detect errors precisely. The error messages should point at the exact error type and error location. The interpreter should not tolerate undeclared variable usage, spelling mistakes of keywords or grammatically ill code.