**The Plus++ Language**

The Plus++ language is a small programming language that has been designed in order to create big integers, increment and decrement them at will and display the result on the screen.

**Lines of Code:** Semi-colon (‘;’) 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 increment statement.
* A decrement statement.
* An output statement.
* A loop statement.

**Comments:**

Comments are written between asterix characters. Can cover multiple lines.

**example:** \*this is a comment\* \* and

this is a comment too\*

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

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

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

3.14159 is not a valid *number* (it is a real number, i.e. float…)

3.0 is not a valid *number* (decimal point and fraction digit should not be displayed)

3. is not a valid *number* (decimal point should not be displayed)

- 5 is not a valid *number* (there should be no blank between the minus sign and the first digit.

--5 is not a valid *number* (only one minus sign allowed)

+5 is not a valid *number* (the unnecessary plus sign is not allowed)

**Variables:** All variables should be declared as a *number*. 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:** number myVar;

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

**Assignment Statement:**

<variable> := <int\_value> ;

**example:** myVar:= 25; \*assigns 25 to myVar\*

yourVar := myVar; \*assigns myVar to yourVar\*

**Increment Statement:** Increase the value of a number.

<variable> += <int\_value> ;

Increments the variable by int\_value.

**example:** sum+=2; \*The value of sum increases by 2\*

**Decrement Statement:** Decrease the value of a number.

<variable> -= <int\_value>;

**example:** total-=t; \*The value of total decreases by t\*

**Write Statement:** Display on screen

write <output\_list>; \*List elements are separated by the keyword “and”\*

The `write` statement is used to output values and strings to the standard output.

**example:** write “The result is:” and 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 the “and” keyword.**

<out\_list>→<out\_list>and<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!”

**The `newline` Keyword:** The `newline` keyword, when used in a `write` statement's output list, instructs the interpreter to advance the output cursor to the beginning of the next line. It does not print any visible characters itself but controls the formatting of subsequent output.

**example:** write "First line." and newline and "Second line.";

\*This would output: First line.

Second line. \*

**Loop:**

repeat <int\_value> times <line>

\*OR\*

repeat <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:**

repeat 10 times write “x”; \*writes 10 x on the screen\*

**Code Block:**

A code block is a list of code lines between curly braces. 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:**

number size;

number sum;

size:= 5;

repeat size times

{ write size and newline;

sum+=size;

}

write newline and “Sum:” and sum;

**Output is:**

5

4

3

2

1

Sum:15

**Errors:** Error detection is an important aspect of a successful Plus++ 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.

**Project 1: Writing a lexical analyzer for the Plus++ Language**

**Due:** **30.05.2025** (with 5% cumulative penalty for each extra day)

The lexical analyzer for the Plus++ 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 “*.plus”* .

example: The command c:\> la myscript

must load the script file called myscript.plus 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)

number size;

number sum;

size:=5;

repeat size times \*ignore me, I am a comment\*

{ write size and newline; \*print to screen\*

sum+=size;

}

write newline and “Sum:” and sum;

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

Keyword(number)

Identifier(size)

EndOfLine

Keyword(number)

Identifier(sum)

EndOfLine

Identifier(size)

Operator(:=)

IntConstant(5)

EndOfLine

Keyword(repeat)

Identifier(size)

Keyword(times)

OpenBlock

Keyword(write)

Identifier(size)

Keyword(and)

Keyword(newline)

EndOfLine

Identifier(sum)

Operator(+=)

Identifier(size)

EndOfLine

CloseBlock

Keyword(write)

Keyword(newline)

Keyword(and)

StringConstant(“Sum:”)

Keyword(and)

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 the 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 executable codes separately.