# Computer Science CS321 - Advanced Programming Techniques
# Bishop's University, Winter 2019
# Instructor: D. Voulioutis

## *ASSIGNMENT 2 The Interpreter*

**Due: Wednesday 27 Feb 2019**

**Initial specifications**

The goal of this assignment is to give you practice with the various types of classes in a setting where they are particularly useful when it comes to extensions and modifications to a project

You will write an interpreter for a machine language – let's call it TML. ( **T**iny **M**achine **L**anguage ) . An interpreter is a program that inputs a source program and directly executes it.

The general form of a machine language instruction is

**label instruction register-list**

***label*** is the label for the line. A string that identifies the position of a particular instruction of the program. This will be used when other instructions want to "jump" to the instruction at that label.

***instruction*** is the string that denotes the kind of instruction. f.e. add, multiply, etc. In our first version of TML, there are instructions for the arithmetic operators (such as add, mul, div sub ), for storing and retrieving integers ( sto ) , and for conditionally branching to other labels ( bnz).

***register-list*** is the list of registers that the instruction manipulates. Registers are simple, integer, storage areas in computer memory, much like variables. In TML, there are 32 registers, numbered 0, 1, ..., 31.

For the time being assume that TML has only the following instructions:

- **L1 add r s1 s2** Add the contents of registers s1 and s2 and store the result in register r.
- **L1 sub r s1 s2** Subtract the contents of register s2 from the contents of s1 and store the result in register r.
- **L1 mul r s1 s2** Multiply the contents of registers s1 and s2 and store the result in register r.
- **L1 div r s1 s2** Divide (Java integer division) the contents of register s1 by the contents of register s2 and store the result in register r.
- **L1 out s1** Print the contents of register s1 on the Java console (using println).
- **L1 sto r x** Store integer x in register r.
- **L1 bnz s1 L2** If the contents of register s1 is not zero, then make the statement labeled L2 the next one to execute.

To begin with we keep the number of different instructions small so that you would have less work to do. For example, there could have been other branch instructions, a negation instruction, an input instruction, and so on. ***The idea is to implement this first set in a way that it will be easy to add more instructions later.*** So think about the design of the classes you will need.

In the above list of instructions:

**L1** is any identifier -- actually, any sequence of non-whitespace characters. Each statement of a program must be labeled with a different identifier.

Each of **s1, s2**, and **r** is an integer in the range 0..31 and refers to one of the 32 registers in the machine that executes language TML.

Here is an example of a TML program to compute factorial of 6. Note that adjacent fields of an instruction (label, opcode, and operands) are separated by whitespace.

```
f0  sto 20 6
f1  sto 21 1
f2  sto 22 1
f3  mul 21 21 20
f4  sub 20 20 22
f5  bnz 20 f3
f6  out 21
```

Instructions of a program are executed in order (starting with the first one), *unless the order is changed by execution of a **bnz** instruction.* Execution terminates when its last instruction has been executed (and doesn't change the order of execution).

Your interpreter should

1. Open a text file that contains the program one instruction per line,
2. Read the program from the file and translate it into an internal form,
3. Print the program,
4. Execute the program, and
5. Print the final value of the registers.

Intentionally the description is very high level with only the functional requirements stated.

This program is an exercise in using *encapsulation*, *abstraction*, the *inheritance* mechanism and object communication between classes in such a way as to ensure ***modifiability*** to the maximum.

We will also use this case study later on to see how the application of certain software design patterns ( the command and visitor pattern ) helps in making the program easy to modify.

It will essentially be marked on correctness ( of course ! ) but also on the ability to incorporate changes in the requirements.

An extension to the assignment ( coming up next as a lab ) will involve changes in the *format, the instructions of the TML, the method of getting the file name from the user and/or other modifications in the specifications* .
Can your program be modified easily enough to support them ??
Can it be modified without having access to the major parts of the source code ( i.e. if you have a compiled Machine class, will it be able to execute the new instructions without modifications ? This involves putting things that may change in different classes from the ones that do not change.

Submit the zipped project folder named as usual