This assignment is to be completed individually

Assignment One
Simple Machine Language (SML)

Software Design and Programming and
Software and Programming III

Spring Term 2016

The sample code mentioned in the text can be found on the module repository under the `SML` project. This coursework can be completed in Java or Scala **only**. The sample code folder for the Scala version is `SML_Scala`.

The aim of this assignment is to give you practice with subclasses, modifying existing code, and the use of reflection.

The problem

You will write an interpreter for a simple machine language — `SML`. The general form of a machine language instruction is

```
label instruction register-list
```

where

`label` is the label for the line.
: Other instructions might "jump" to that label.

`instruction` is the actual instruction.
: In SML, there are instructions for adding, multiplying and so on, for storing and retrieving integers, and for conditionally branching to other labels (like an *if* statement).

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

SML has 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 lin 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

where

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 SML.

Here is an example of an SML program to compute factorial 6.

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

Note that adjacent fields of an instruction (label, opcode, and operands) are separated by whitespace.

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 will:

1. Read the the name of a file that contains the program from the command line (via `String[] args`),
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.

This looks like a tall order, but have no fear; we provide you with some of the code, so that you can concentrate on the interesting use of subclasses and reflection. Follow our directions on doing the project, and the project should take 3-5 hours at most, once you have read and understood the code (and this document).

Design of the program

We provide some of the classes, provide specifications of a few, and leave a few for you to write/complete. The code we provide does some of the *dirty work* of reading in a program and translating it to an internal form, so you can concentrate just on the code that executes the program. Study the class `Machine` first, for it is the heart of the program.

Please note: The following description mainly relates to the Java version of the source code. The Scala version is somewhat shorter as several constructs are easier to achieve using that language. Please consult the Scala source code for further details.

Studying the program

You are provided with some skeleton code which is on the module repository.

Look at the fields of class `Machine`, which contain exactly what is needed to execute an SML program:

- the *labels* defined in the program,
- the program itself in an *internal form*,
- the *registers* of the machine, and
- the *program counter* — the number of the next instruction to execute.

Array like structures are used for the labels and the instructions of the machine because there is no limit to the size of an SML program. An array is used for the registers because there are always exactly 32 registers.

Now read method `Machine.execute`, which executes the program. It is a typical *fetch-execute cycle* that all machines have in some form. At each iteration, the instruction to execute is fetched, the program counter is incremented, and the instruction is executed. The order of the last two instructions is important, because an instruction (e.g. `bnz`) might change the program counter.

The class `Translator` contains the methods that read in the program and translate it into an internal form. Very little error checking goes on here. For example, there is no checking for duplicate label definitions, for the use of a label that doesn't exist, and for a register number not in the range $0 \ldots 31$.

Finally, study the `main` method.

Class `Instruction` and it subclasses

All the programming that you do has to do with class `Instruction` and its subclasses. The specification of class `Instruction` has been given to you — open the file

```
Instruction.java
```

and examine it. This class is *abstract*, because it should not be instantiated. The method `execute` is also *abstract*, forcing every subclass to implement it (we have avoided the use of Java 8 features in this coursework).

Every instruction has a *label* and an *operation* — that is exactly what is common to every instruction. Therefore, these properties are maintained in the base class of all instructions.

- Your first task is to complete the methods in class `Instruction` — this may require you to add some fields, which should be *protected*, so that they are accessible in subclasses.

- Now create a subclass of `Instruction` for each kind of SML instruction and fix method `Translator.instruction` so that it properly translates that kind of instruction.

  Recommended: write one instruction at a time and check it out thoroughly, before proceeding to the next!

  For example, your first program will consist only of *add* instructions. After you have checked that the instruction add works correctly, progress to working with the *add* and *subtract* instructions, etc. As you do this, you will see that each successive class can be written by duplicating a previous one and modifying it (obviously avoiding too much repeated code).

- For each instruction, the subclass needs appropriate fields, a constructor, method `toString`, and a method `execute`; `toString` and `execute` should override the same methods in class `Instruction` using appropriate annotations.

- Start with the `add` instruction, because the code for translating it is already there — in method `Translator.instruction`. Initially, the program will not compile because there is no class for the instruction `add`. Once that class is suitably written, the program will compile.

- After you finish writing a subclass for an SML (except for `add`), you have to add code to the method `Translator.instruction` to translate that instruction. The code for translating `add` should help you with this.

Final requirement — reflection

Now take the `switch` statement that decides which type of instruction is created, modify the code so that it uses *Java reflection* to create the instances, i.e., remove the explicit calls to the subclasses. This will allow the SML language to be extended without having to modify the original code.

Submission

As part of your portfolio your repository will be cloned at the appropriate due date and time.