Simplified Virtual Machine
INFSEN02-1, INFSEN22-1

The goal of this assignment is to realise a virtual machine, called *Simplified Virtual Machine* (SVM) for a fictional assembly language (*Assembly SVM*). This documents describes the specifications of the SVM and its assembly language. You will be provided with a template for an F# solution containing a pre-built parser for the assembly language. The parser does the syntax analysis on a program written in SVM Assembly and outputs a data structure representing the program itself, which is described below. Your task is to build the interpreter, which is the component of the virtual machine that executes each statement in the assembly language and alters the state of the memory of the virtual machine. Your implementation must follow the following constraints:

- You cannot change in any way the Parser (and thus the language specifications) of Assembly SVM.
- You cannot change the semantics (how the statements work) of Assembly SVM.
- You cannot use ANY imperative construct/statement other than commands to print on the shell. This means, for instance, no mutable variables, no loops, no mutable data structures, etc.
- If you want to use any language other than F# you must implement a parser for Assembly SVM on your own using the same specifications.

1 Architecture of the SVM

The SVM is made of a set of memory locations, which index is called address. The total amount of memory locations available in the SVM can be specified by the user. Each memory location can contain integer, floating point, or string values (for simplicity, as this is different from hardware architectures and standard virtual machines where you only save bytes). Furthermore, unlike normal virtual machines, the values used in the assembly language are typed, meaning that the type compatibility is checked when executing a statement, in the same fashion as in high-level programming languages where, for example, you cannot sum a `string` with an `integer`.

The virtual machine has also 4 registers `reg1`, `reg2`, `reg3`, and `reg4`, that are special memory location used by the instructions to save temporary results or to contain temporary data. The data types contained in the registers can be the same as the main memory. It has also a special register, called *Program Counter*, which contains the index of the current instruction to be executed.

Unlike low-level architectures, the program you have to execute is not saved in the memory but it is kept separate (thus the memory contains only the data and not the instructions the SVM has to execute).

2 Interface with the Parser

The Parser is defined in the Project `SVMParser` in the template solution. Please do not alter in any way this project otherwise the template will not work properly. The data structures you have to use to interface with the parser are defined in the file `AST.fs`. Import the module `SVMAST` after adding a reference to the `SVMParser` in the project you use for your own implementation to use the data structures. The parser processes the source file you want to load and creates a `Program` data structures, which is a list containing `Instructions`. The `Instruction` type is a discriminate union containing the definition of all the possible statements in Assembly SVM. Each `Instruction` can accept one or more arguments. Each `Argument` can be a literal (value) of type integer, floating point, or string, a `Register`, or a memory address. Some operators will accept only a `Register` as left argument (see the instruction specification below). `Registers` are represented as a discriminated union where each case represents the register itself.

3 Syntax of Assembly SVM

A program in Assembly SVM consists of one or more lines, each one containing a **single statement** in Assembly SVM. Thus, different statements must be written in different lines of the program. You can comment part of your program with the same syntax of Java/C++/C# languages, that is // for single line comments and /* */ for multi-line comments. Also empty lines and white spaces are simply ignored by the parser, like in traditional programming languages. The language has three main categories of instructions:

- *Memory manipulation*: this statement takes two argments, one being the memory location/register to which you want to copy the data, and one being the data, memory location, or register you want to copy from.
- *Operators*: they might be unary (with one argument) or binary (with two arguments).
- *Labels*: They are identifiers that can be use to reference specific points of the program. To define a labels simply use the symbol `#` followed by a variable name, which can be any alphanumerical sequence of characters starting with a letter. They can also contain the symbol '_'. For example `#hello_world` is a valid label name, while `#090error` is not a valid label name.
- *Jump statements*: they alter the program counter by setting it to the line where a specific label is defined, thus allowing to implement the equivalent of `if-then-else` and `loops` of imperative programming.

The argument of an instruction can have three different types:

- *Literals*: they are constant values like numbers (`10`, `1053.23`) or strings (`"Hello world!"`).
- *Memory addresses*: they are defined as integer numbers enclosed by square brackets, like `[100]`, which means "the memory location at index 100".
- *Registers*: they are simply denoted with the keywords `reg1`, `reg2`, `reg3`, and `reg4`.
- *Register references*: they are used when a register contains an integer number representing the address of a memory location, rather than a value. It is written just by enclosing the register keyword with square brackets. For instance, `[reg1]` means "the content of the memory address stored in `reg1`. Thus, if the register contains the value 1000, this will use the content of the memory location at 1000.

4 Instruction Behaviours

When the SVM execute a statement, it performs 3 steps:

- Check if the operands are type-compatible (according to the definition below). If they are not, it throws a runtime error.
- If the operand test succeeds, then it executes the instruction behaviour (see the table below).
- After executing the instruction, it increases the program counter by 1 **only if the instruction behaviour does not change the program counter, namely for the jump instructions**.

Below you find the details of all the operators accepted by the SVM language. Note that an argument which is an address can be either a *Memory address* or a `Register reference`, as described above.

Name	**Arg1**	**Arg2**	**Behaviour**
NOP	None	None	This statement does not do anything and it leaves the state of the SVM unchanged.
MOV	Address or Register	Register, Address, or Constant	Copy the content or value of Arg2 into Arg1. If the memory address is out of range then it throws a runtime exception.
AND	Register	Register, Address, or Constant	If both arguments are >= 0 then it stores 1 into Arg1, otherwise -1. It accepts only integer numbers, otherwise it raises a runtime exception.
OR	Register	Register, Address, or Constant	If both arguments are < 0 then it stores -1 into Arg1, otherwise 1. It accepts only integer numbers, otherwise it raises a runtime exception.

NOT	Register	None	If the argument is non-negative, it stores -1 in Arg1, otherwise it stores 0. It accepts only an integer number, otherwise it raises a runtime exception.
MOD	Register	Register, Address, or Constant	It computes the modulus operation (remainder of the integer division) with Arg1 and Arg2 and stores the result in Arg1. It accepts only numerical arguments (integer or float), otherwise it raises a runtime exception.
ADD	Register	Register, Address, or Constant	It computes the sum operation with Arg1 and Arg2 and stores the result in Arg1. It accepts only numerical arguments (integer or float), otherwise it raises a runtime exception.
SUB	Register	Register, Address, or Constant	It computes the difference operation with Arg1 and Arg2 and stores the result in Arg1. It accepts only numerical arguments (integer or float), otherwise it raises a runtime exception.
MUL	Register	Register, Address, or Constant	It computes the multiplication operation with Arg1 and Arg2 and stores the result in Arg1. It accepts only numerical arguments (integer or float), otherwise it raises a runtime exception.
DIV	Register	Register, Address, or Constant	It computes the division operation with Arg1 and Arg2 and stores the result in Arg1. Note that with integers you do the integer division and with floats the floating point division. It accepts only numerical arguments (integer or float), otherwise it raises a runtime exception.
CMP	Register	Register, Address, or Constant	It compares the two arguments, returning -1 if Arg1 $<$ Arg2, 0 if they are equal, 1 if Arg1 $>$ Arg2. The result is stored in Arg1. It accepts only numerical values otherwise it raises a runtime exception.

#	Id	None	The operator defines a label with the name specified in Arg1. It is not allowed to define labels with the same name, so in this case a runtime error should be raised.
JMP	Id	None	Jump at the point of the program where the label with the name in Arg1 is defined. This instruction modifies the program counter to do so.
JC	Id	Register	Jump at the point of the program where the label with the name in Arg1 is defined, if Arg2 is >= 0. This instruction modifies the program counter to do so.
JEQ	Id	Register	Jump at the point of the program where the label with the name in Arg1 is defined, if Arg2 is 0. This instruction modifies the program counter to do so.

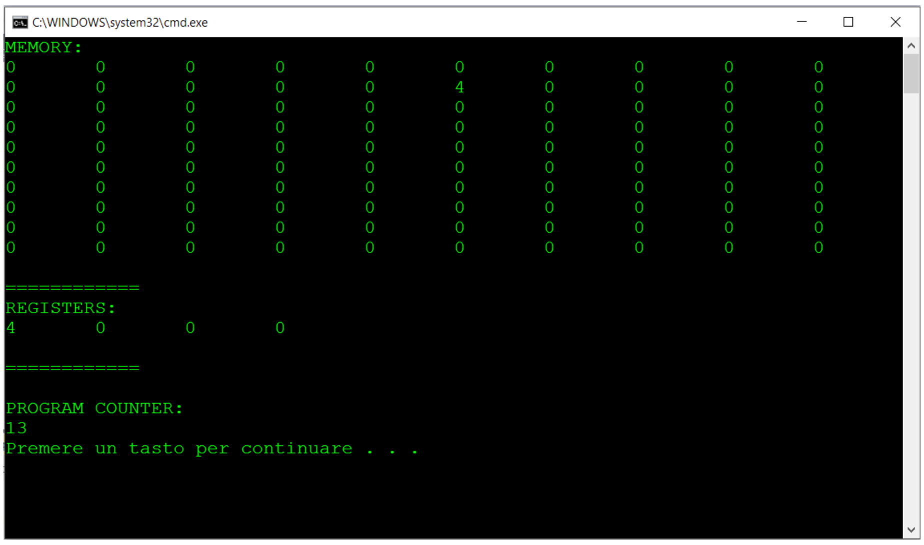
A further note on JC and JEQ: you use these instructions in combination with the CMP instruction. First you run CMP to compare the values, and then you use its result in the JC or JEQ instruction.

Example: Consider the program

```
if (x > 0)
{
  x -= 1;
}
else
{
  x += 1
}
```

this will be translated in Assembly SVM, assuming that the value of `x` is stored at the memory address 15, as:

```
//move the content of address 15 in reg1
mov reg1 [15]
//move the constant 0 in reg2
mov reg2 0
//compare reg1 with reg2
cmp reg1 reg2
//negate the result (we want to jump if x <= 0)
not reg1
jc else reg1
//move back x into reg1 (it has been overwritten)
mov reg1 [15]
// x -= 1
sub reg1 1
```



```
C:\WINDOWS\system32\cmd.exe
MEMORY:
0        0        0        0        0        0        0        0        0        0
0        0        0        0        0        4        0        0        0        0
0        0        0        0        0        0        0        0        0        0
0        0        0        0        0        0        0        0        0        0
0        0        0        0        0        0        0        0        0        0
0        0        0        0        0        0        0        0        0        0
0        0        0        0        0        0        0        0        0        0
0        0        0        0        0        0        0        0        0        0
0        0        0        0        0        0        0        0        0        0
0        0        0        0        0        0        0        0        0        0

============
REGISTERS:
4        0        0        0

============

PROGRAM COUNTER:
13
Premere un tasto per continuare . . .
```

Figure 1: Memory dump

```
//skip the else block
jmp endif
#else
// x += 1
add reg1 1
#endif
//update the variable
mov [15] reg1
```

5 Assignment

Your task is to build the interpreter for Assembly SVM. This means that the template will generate a data structure representing the program, as defined in the module `SVMAST`, and you will have to build a software that executes that code, according to the instruction set defined above. Your program should be able to accept as input the source file written in Assembly SVM and the size (number of addresses) of the memory of the SVM you want to run the program on. The interpreter should raise an exception (which must be captured appropriately) in case of incompatible arguments, as described in the instruction behaviour. The output of the program should be a *dump* of the SVM, which contains a visualization of the memory, the registers, and the program counter. Organize the memory display as a table with the memory locations content with 10 items per row. An example of a memory dump is shown in Figure 1.
Your implementation must observe the following rules:

- You are not allowed to use any imperative constructs, other than statements that print on the standard output (such as `printfn`) and exceptions (`try-with` and `exception` definitions). This means no mutable values (`let mutable`), no variable assignments, such as `let mutable x = 0; x`

`<- x + 1`, no `classes` (although immutable records are allowed), no `for` or `while` loops (you can use *list comprehension* if you understand how they work, but this is out of the scope of the course), no arrays (only lists). You are allowed to use methods and properties in conjunction with immutable records if they behave in an immutable way (so if they respect the constraints above).

- Your implementation should show that you know as much as possible the constructs of functional programming, so use as much as possible (of course where it makes senese) higher order functions, tuples, discriminated unions, immutable records, etc.
- Your solution must be tested with the examples included in the template but you are allowed to add your own programs written in Assembly SVM and present them.
- You are not allowed to change any part of the language definition, that is, the grammar, the syntactical representation generated by the Parser, and the instruction set.

5.1 Evaluation

Below you find how the evaluation matrix

Feature	Maximum score	Details
SVM representation	2	The SVM is structured using a record with appropriate methods, properties, or functions implementing the functionalities (2 points). The SVM is implemented but without an explicit data type (1 point) or functions are not well structured according to the functional paradigm. 0 otherwise
Operators	4	Boolean, arithmetic, and comparison operators are implemented correctly (4 points). Operators are implemented but the exceptions are not handled properly or the argument by reference does not work (2 points). 0 otherwise.
Jump	4	Jump operators and labels are correctly implemented (4 points). Some jump operators are missing or not fully working (2 points). 0 otherwise.