Chapter II

The project

AbstractVM is a machine that uses a stack to compute simple arithmetic expressions. These arithmetic expressions are provided to the machine as basic assembly programs.

II.1 The assembly language

II.1.1 Example

As an example is still better that all the possible explanations in the world, this is an example of an assembly program that your machine will be able to compute:

```
; -------------
; exemple.avm -
; -------------

push int32(42)
push int32(33)

add

push float(44.55)

mul

push double(42.42)
push int32(42)

dump

pop

assert double(42.42)

exit
```

II.1.2 Description

As for any assembly language, the language of `AbstractVM` is composed of a series of instructions, with one instruction per line. However, `AbstractVM`'s assembly language has a limited type system, which is a major difference from other real world assembly languages.

- Comments: Comments start with a `';'` and finish with a newline. A comment can be either at the start of a line, or after an instruction.

- **push *v***: Pushes the value v at the top of the stack. The value v must have one of the following form:
  - **int8(*n*)** : Creates an 8-bit integer with value n.
  - **int16(*n*)** : Creates a 16-bit integer with value n.
  - **int32(*n*)** : Creates a 32-bit integer with value n.
  - **float(*z*)** : Creates a float with value z.
  - **double(*z*)** : Creates a double with value z.

- **pop**: Unstacks the value from the top of the stack. If the stack is empty, the program execution must stop with an error.

- **dump**: Displays each value of the stack, from the most recent one to the oldest one WITHOUT CHANGING the stack. Each value is separated from the next one by a newline.

- **assert *v***: Asserts that the value at the top of the stack is equal to the one passed as parameter for this instruction. If it is not the case, the program execution must stop with an error. The value v has the same form that those passed as parameters to the instruction **push**.

- **add**: Unstacks the first two values on the stack, adds them together and stacks the result. If the number of values on the stack is strictly inferior to 2, the program execution must stop with an error.

- **sub**: Unstacks the first two values on the stack, subtracts them, then stacks the result. If the number of values on the stack is strictly inferior to 2, the program execution must stop with an error.

- **mul**: Unstacks the first two values on the stack, multiplies them, then stacks the result. If the number of values on the stack is strictly inferior to 2, the program execution must stop with an error.

- **div**: Unstacks the first two values on the stack, divides them, then stacks the result. If the number of values on the stack is strictly inferior to 2, the program execution must stop with an error. Moreover, if the divisor is equal to 0, the program execution must stop with an error too. Chatting about floating point values is relevant a this point. If you don't understand why, some will understand. The linked question is an open one, there's no definitive answer.

- **mod**: Unstacks the first two values on the stack, calculates the modulus, then stacks the result. If the number of values on the stack is strictly inferior to 2, the program execution must stop with an error. Moreover, if the divisor is equal to 0, the program execution must stop with an error too. Same note as above about fp values.

- **print**: Asserts that the value at the top of the stack is an 8-bit integer. (If not, see the instruction `assert`), then interprets it as an ASCII value and displays the corresponding character on the standard output.

- **exit**: Terminate the execution of the current program. If this instruction does not appears while all others instruction has been processed, the execution must stop with an error.

`For non commutative operations, consider the stack v1 on v2 on stack_tail, the calculation in infix notation v2 op v1.`

When a computation involves two operands of different types, the value returned has the type of the more precise operand. Please do note that because of the extensibility of the machine, the precision question is not a trivial one. This is covered more in details later in this document.

II.1.3 Grammar

The assembly language of `AbstractVM` is generated from the following grammar (`#` corresponds to the end of the input, not to the character `'#'`):

```
S := INSTR [SEP INSTR]* #

INSTR :=
     push VALUE
   | pop
   | dump
   | assert VALUE
   | add
   | sub
   | mul
   | div
   | mod
   | print
   | exit

VALUE :=
    int8(N)
  | int16(N)
  | int32(N)
  | float(Z)
  | double(Z)

N := [-]?[0..9]+

Z := [-]?[0..9]+.[0..9]+

SEP := '\n'+
```

II.1.4 Errors

When one of the following cases is encountered, `AbstractVM` must raise an exception and stop the execution of the program cleanly. It is forbidden to raise scalar exceptions. Moreover your exception classes must inherit from `std::exception`.

- The assembly program includes one or several lexical errors or syntactic errors.
- An instruction is unknown
- Overflow on a value
- Underflow on a value
- Instruction pop on an empty stack
- Division/modulo by 0
- The program doesn't have an `exit` instruction
- An `assert` instruction is not true
- The stack is composed of strictly less that two values when an arithmetic instruction is executed.

Perhaps there are more errors cases. However, you machine must never crash (segfault, bus error, infinite loop, unhandled exception, ...).

II.1.5 Execution

Your machine must be able to run programs from a file passed as a parameter and from the standard input. When reading from the standard input, the end of the program is indicated by the special symbol ";;" otherwise absent.

Be very careful avoiding conflicts during lexical or syntactic analysis between ";;" (end of a program read from the standard input) and ";" (beginning of a comment.)

Now let see some examples of execution:

```
$>./avm
push int32(2)
push int32(3)
add
assert int32(5)
dump
exit
;;
5
$>
```

```
$>cat sample.avm
; -------------
; sample.avm -
; -------------

push int32(42)
push int32(33)
add
push float(44.55)
mul
push double(42.42)
push int32(42)
dump
pop
assert double(42.42)
exit
$>./avm ./sample.avm
42
42.42
3341.25
$>
```

```
$>./avm
pop
;;
Line 1 : Error : Pop on empty stack
$>
```

The error message is given as an example. Feel free to use yours instead.

Chapter III

Mandatory part

In order to help you with your project, we provide you with the following instructions that you **MUST** respect.

III.1 Generic instructions

- You are free to use any compiler you like.
- Your code must compile with: `-Wall -Wextra -Werror`.
- You are free to use any `C++` version you like.
- You are free to use any library you like.
- You must provide a Makefile with the usual rules.
- Any class that declares at least one attribute must be written in canonical form. Inheriting from a class that declares attributes does not count as declaring attributes.
- It's forbidden to implement any function in a header file, except for templates and the virtual destructor of a base class.
- The "keyword" `"using namespace"` is forbidden.

III.2 The IOperand interface

`AbstractVM` uses 5 operand classes that you must declare and define:

- **Int8** : Representation of a signed integer coded on 8bits.
- **Int16** : Representation of a signed integer coded on 16bits.
- **Int32** : Representation of a signed integer coded on 32bits.
- **Float** : Representation of a float.
- **Double** : Representation of a double.

Each one of these operand classes **MUST** implement the following `IOperand` interface:

```
class IOperand {

public:

    virtual int           getPrecision( void ) const = 0;   // Precision of the type of the instance
    virtual eOperandType  getType( void ) const = 0;        // Type of the instance

    virtual IOperand const *  operator+( IOperand const & rhs ) const = 0; // Sum
    virtual IOperand const *  operator-( IOperand const & rhs ) const = 0; // Difference
    virtual IOperand const *  operator*( IOperand const & rhs ) const = 0; // Product
    virtual IOperand const *  operator/( IOperand const & rhs ) const = 0; // Quotient
    virtual IOperand const *  operator%( IOperand const & rhs ) const = 0; // Modulo

    virtual std::string const & toString( void ) const = 0; // String representation of the instance

    virtual ~IOperand( void ) {}

};
```

Considering similarities between operand classes, it can be relevant to use a class template. However, this is not mandatory.

III.3 Creation of a new operand

New operands **MUST** be created via a "factory method". Search `Google` if you don't know what it is. This member function must have the following prototype:

```
IOperand const * createOperand( eOperandType type, std::string const & value ) const;
```

The `eOperandType` type is an `enum` defining the following values: `Int8`, `Int16`, `Int32`, `Float` and `Double`.

Depending on the `enum` value passed as a parameter, the member function `createOperand` creates a new `IOperand` by calling one of the following `private` member functions:

```
IOperand const * createInt8( std::string const & value ) const;
IOperand const * createInt16( std::string const & value ) const;
IOperand const * createInt32( std::string const & value ) const;
IOperand const * createFloat( std::string const & value ) const;
IOperand const * createDouble( std::string const & value ) const;
```

In order to choose the right member function for the creation of the new `IOperand`, you **MUST** create and use an array (here, a `vector` shows little interest) of pointers on member functions with enum values as index.

III.4 The precision

When an operation happens between two operands of the same type, there is no problem. However, what about when the types are different ? The usual method is to order types using their precision. For this machine you should use the following order: `Int8 < Int16 < Int32 < Float < Double`. This order may be represented as an `enum`, as `enum` values evaluate to integers.

The pure method `getPrecision` from the interface `IOperand` allows to get the precision of an operand. When an operation uses two operands from two different types, the comparison of theirs precisions allows to figure out the result type of the operation.

III.5 The Stack

AbstractVM is a stack based virtual machine. Whereas the stack is an actual stack or another container that behaves like a stack is up to you. Whatever the container, it `MUST` only contain pointers to the abstract type `IOperand`.

Chapter IV

Bonus part

Once you're done with the mandatory part and sure you'll get the maximum grade for it, consider doing the following bonuses :

- Making your VM capable of diagnosing all errors in a file without stopping at the first error encountered.
- Having a well structured lexer/parser couple, with clearly defined roles.

You can add some bonuses of your own creation if you like, but most of the bonus points will be reserved for the above bonuses.