Project 1 : Abacus

Submission instructions

At the end of the practical, your Git repository must follow this architecture:

```
firstname.lastname-abacus-2026/
|-- README
|-- .gitignore
|-- Abacus.sln
|-- Abacus/
    |-- Everything except bin/ and obj/
    |-- Abacus.csproj
    |-- Program.cs
```

Do not forget to check the following requirements before submitting your work:

- You shall obviously replace `firstname.lastname` with your login.
- The `README` file is mandatory.
- There must be no `bin` or `obj` folder in the repository.
- Remove all personal tests from your code.
- **The code MUST compile ! Otherwise you will not be graded.**

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

1 Introduction

1.1 Objectives

Until now, you have been guided throughout to write your functions within an already provided structure. The objective of this tutorial is to let you think about the architecture of a project and how to articulate the different elements together. Don't worry, your ACDC will be there to advise you on the structure to adopt and, as always, to answer your questions.

1.2 Unfolding

This practical will be a two-week project. More details will be given in the course and the exercises. It is, therefore, essential to read everything carefully.

As usual, the handout will be done with git. As the project will be evaluated on the output of your program, the respect of the specifications and the expected outputs is essential. An intermediate submission will be made available to you to help you detect any errors.

2 Course

The core of the project will be the evaluation of arithmetic expressions. We focus for now on the essential elements to build it. It is therefore strongly recommended to make sure you understand everything that will be explained in this section before you start writing code.

Tip

Don't forget that your ACDCs are here to answer your questions.

2.1 Lexical analysis

Your program will take as input a string given by the user. The problem is that this format is not suitable for further processing. It is indeed easier to read a list of atomic symbols[1] rather than characters that only make sense in a larger context.

Without lexical analysis, we have to read each character one by one, stop when we detect the end of a number or an operator, and then perform the subsequent operations on it. We end up in a situation where we try to perform many operations at the same time while it would be better to separate them. This is the purpose of lexical analysis, which consists in transforming a string of characters into a sequence of `tokens`.

Consider the following example:

$$42 + 1337 \times 11$$

The result of the lexical analysis could be a sequence of tokens like:

Number(`42`)	Operator(`+`)	Number(`1337`)	Operator(`*`)	Number(`20`)

Once the lexical analysis is done, it is much easier to process the input. Thus, although this step is not absolutely mandatory, it is highly recommended not to ignore it.

[1]Called tokens

2.2 Syntactic analysis

The lexical analysis consists in converting the stream of `tokens` into a data structure allowing the programmer to easily manipulate the expression.

2.2.1 Abstract syntax tree (AST)

A tree structure lends itself particularly well to the representation of arithmetic expressions as you have seen in your MiMos.

In this section, we limit ourselves, without loss of generality on expressions, to binary operators only. A syntax tree is then a binary tree whose internal nodes are the operators and whose leaves are the operands.

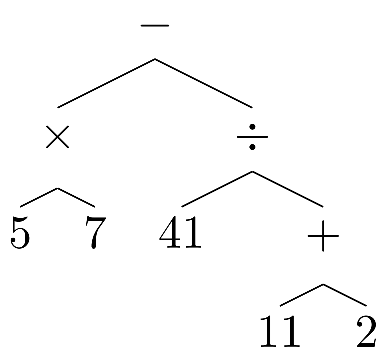
Operators — Definitions

- **Operators** represent actions to be performed on **operands**. The most classical arithmetic operators are $+$, $-$, $\times$ and $\div$. In a mathematical expression, the operands are the numbers.
- The **precedence** of an operator is the relative priority of one operator with respect to another. For example, *times* and *div* have the same precedence. The latter is superior to $+$ and $-$ which in turn have the same precedence.
- The **associativity** of an operator is the way it is possible to parenthesize the operator without changing the meaning of the expression. If a, b and c are operands, the operator $\star$ is said to be *left associative* if the expression $a \star b \star c$ has the same meaning as $(a \star b) \star c$
- The **arity** of an operator is the number of operands that this operator accepts. The arithmetic operators presented above are all binary operators, that is, they have an arity of 2.

Consider the expression below:

$$5 \times 7 - 41 \div (11 + 2)$$

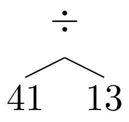
It can be represented by the following AST:



We note that the internal nodes correspond to the operators and the leaves to the operands.

Let's break down the evaluation of the AST. We first evaluate the left subtree. The operation performed is 5×7, namely 35.

In the right subtree, we recursively evaluate the right *sub*-subtree and perform the operation $11+2$. The right subtree is thus reduced:



This tree has to be evaluated again; so we perform the operation $41 \div 13$, meaning 3. Finally, we have to perform the operation $35 - 3$, which is 32.

2.2.2 Parsing

Parsing consists in building a syntax tree from a stream of `tokens`. Different algorithms exist for this, one of which is called Shunting-yard. We encourage you to use it.

3 Assignement

This project is about implementing an integer arithmetical expression calculator. This calculator is divided into three levels of increasing difficulties :

- Evaluating an expression in Reverse Polish Notation (RPN).
- Evaluating a parenthesised arithmetical expression.
- Some diverse bonuses including functions and implicit multiplication.

In this project you are allowed every function of the C# standard library except the functions that allow the evaluation of arithmetical expressions. Using such functions will be considered as cheating and will be severely punished.

Your program should read the content of the standard input stream[2] and evaluate the expressions given.

You program should also take an optional parameter `--rpn` that will specify that you are expecting an expression in Reverse Polish notation. For instance the program should behave as follows:

```
42sh> echo '2 3 + 2 *' | ./abacus --rpn
10
42sh> echo $?
0
42sh> echo '2 3 + 2' | ./abacus --rpn
Syntax Error: operator expected.
42sh> echo $?
2
42sh> echo '2 3 + 2' | ./abacus --expression
Unknown Argument: 'expression'
42sh> echo $?
1
```

In the standard output, we expect the result of the evaluation if no errors have occured. If there is an error, you should print a meaningful error message on the standard error stream[3].

The environment variable `?` contains the exit code of your program. Thus, if you wish to show the exit code of the program you have just run in your shell, you can use the command: `echo $?`.

Your program should have the following exit codes:

- 0 if the evaluation exited successfully.
- 1 if a given argument is not given or is invalid.
- 2 if an error has occurred (Syntax error, parenthesis error, etc)
- 3 if an arithmetic exception has occurred (division by zero or unbound variables)

Be careful, the exit code and the content of the standard output are two different things. You can look up the documentation of `System.Environment.Exit` or the documentation on the return value of the `Main` function to know how to use exit codes.

You are free to print whichever error message you please in the error output but there must be something printed in case of a problem.

If you have any doubts regarding the behavior of your program, please refer to the reference binary that is given with the subject. Your program must have the EXACT same behavior[4].

[2] or stdin
[3] see Console.Error.WriteLine
[4] excepted for the error message which are up to you.

Part 1: Reverse Polish Notation (rpn)

The first part is about evaluating expressions in reverse polish notation. There is no unary $-$ and $+$ in RPN, so you have to handle only binary operators so far :

- The operator $+$
- The operator $*$
- The operator $/$
- The operator $-$
- The operator $\wedge$
- The operator $\%$

A few examples :

```
42sh> echo '2 3 + 2 *' | ./abacus --rpn
10
42sh> echo '2 3 + 2' | ./abacus --rpn
Syntax Error: operator expected.
42sh> echo $?
2
42sh> echo '0 2 * 1 + 3 -' | ./abacus --rpn
-2
42sh> echo '2 0 /' | ./abacus --rpn
Runtime Exception: Division by zero.
42sh> echo $?
3
42sh> echo '2 2 ^' | ./abacus --rpn
4
```

Part 2: Arithmetical expressions

Same thing than for the RPN but you have to handle operator priority, unary operators as well as parentheses. If parentheses are not balanced, you should raise a Syntax Error.

A few examples :

```
42sh> echo '2 + 3*4' | ./abacus
14
42sh> echo '(2 + 3 * 4' | ./abacus
Syntax Error: Unbalanced parentheses
42sh> echo $?
2
42sh> echo '(2 + 1) ^ 5 * 2' | ./abacus
486
42sh> echo '-1 * 2' | ./abacus
-2
```

Part 3.1: Functions

You must implement the following functions :

- `sqrt(a)` that returns the square root rounded to the closest integer.
- `max(a, b)` that returns the maximum of the two integers.
- `min(a, b)` that returns the minimum of the two integers.
- `facto(a)` that returns the factorial of this number.
- `isprime(a)` that returns 1 if the number is a prime number, 0 otherwise.
- `fibo(a)` that returns a-ith Fibonnacci number.
- `gcd(a, b)` that return the GCD of a and b.

Be careful about the arithmetical exceptions that these functions add and don't forget to add them in RPN as well. For instance :

```
42sh> echo '2 3 max isprime' | ./abacus --rpn
1
```

3.1 Part 3.2: Variables and implicit multiplication

The objective of this part is to manage the use of variables. A valid variable name is any string of characters beginning with a letter or an underscore ("_") followed by a (potentially empty) sequence of letters, digits and underscores [5]. The names of the functions defined in the previous section are reserved and, therefore, cannot be used as variable names.

You must implement the operator `=` which takes as left operand the name of a variable and as right operand a valid expression to assign to the variable. This operator returns the result of the evaluation of the right operand.

Example:

```
42sh> echo 'a = b = 2' | ./abacus
2
42sh> echo 'a b 2 = =' | ./abacus --rpn
2
```

We also add the possibility to separate several expressions with a semicolon. In this case, the variables are preserved from one expression to the next and the value to be displayed is that of the last expression. It should be noted that the semicolon is not an operator and its use is the same in both RPN and infix form.

Finally, you will need to implement implicit multiplication for parentheses and variables.

Example:

```
42sh> echo 'a = 2(3 + 4); b = 5 + 2a; b = a * b; b / 2' | ./abacus
231
```

[5]This corresponds to the regular expression `[A-Za-z_][A-Za-z0-9_]*`.

Part 4: Bonuses

These bonuses are not graded but they are, nonetheless, very interesting to explore. You can pick your own bonuses (and mention them in your README) or choose from the list below [6]. Be careful not to alter the behavior of the program. If you wish to do so, you can use a special flag that will not be tested `--additionals`. Otherwise, your program should behave EXACTLY like the reference.

Here are a few ideas of bonuses:

- A REPL mode (`--additionals=REPL`).
- Vectorial operations (addition, dot product, vectorial product).
- Matrix operations .
- The ability to combine multiple bonuses:
  (`--additionals=REPL,VECTORS,MATRIX`) for instance.

There is nothing more deceptive,
than an obvious fact.

[6] Or do both!