TP C#12: One parser to rule them all

Assignment

Archive

You must submit a zip file with the following architecture :

```
rendu-tp-firstname.lastname.zip
|-- firstname.lastname/
    |-- AUTHORS
    |-- README
    |-- TPCS12/
        |-- TPCS12.sln
        |-- TPCS12/
            |-- EvalExpr/
                |-- Everything except bin/ and obj/
            |-- List/
                |-- Everything except bin/ and obj/
```

- You shall obviously replace *firstname.lastname* with your login (the format of which usually is *firstname.lastname*).
- The code **MUST** compile.
- In this practical, you are allowed to implement methods other than those specified. They will be considered bonuses. However, you must keep the archive's size as small as possible.

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline.
Here is an example (where `$` represents the newline and ␣ a blank space):

```
*␣firstname.lastname$
```

Please note that the filename is AUTHORS with **NO** extension. To create your `AUTHORS` file simply, you can type the following command in a terminal:

```
echo "* firstname.lastname" > AUTHORS
```

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

In this TP you will see two important ideas in programming.
The **generics** will allow you to code some project that can be easily reused in other projects. You will be able to make generic code.
A parser: You have already done some simple evaluations, and probably encountered some issues on Mathmageddon and TinyBistro. Now it's time to mix everything to see how a mathematical expression is evaluated!

important

Both parts of this TP are totally independent. It's not necessary to finish the first to be able to do the second.

If you need any information on the different data structures of this TP: `https://ionisx.com`

2 Course

2.1 The generics

The **generics** are a very useful and powerful notion in C# (and some other languages). With this feature, we can create functions and classes that can accept any type (as long as the operations applied on the generic object can accept its type). This functionality allows us to factorise our code when we want to apply the same behavior to several different types.

Here is an example:

```
1  //<T> tell that the function is generic, T is a type.
2  void print_line<T>(T elt)
3  {
4      Console.WriteLine(elt);
5  }
6  //We use it like this:
7  print_line<int>(3);
8  //The T of the generic is replaced by int, so the function knows what to do.
9  //To use our function with a string we can do:
10 print_line<string>("These violent deadlines have violent ends.");
```

If you want more informations, you can find the documentation here: `https://msdn.microsoft.com/en-us/library/ms379564(v=vs.80).aspx`

2.2 Let us parse

Parsing is a notion as important as it is undervalued in computer science. You have already done some of this with a simple evaluation as in Brainfuck and data sharing with JSON. You will discover that it is not as easy as it has been previously.
In Brainfuck, you had to interpret a simple language that is read from left to right and where each symbol had only one meaning. The question is : What do we do with expressions that do not evaluate from left to right and where a symbol can have multiple meanings?

2.2.1 A parser : how does it works ?

A **parser** has three main categories :

- the **Lexer**
  Its job is to convert the different sequence of characers in easily usable data for us, we will name them **tokens**.
  A **token** we be represented by a class that contains :
  - An *enum* that corresponds to the lexical entity
  - A *string* that contains the part that corresponds to the lexical entity in the expression we have to parse

  Thus, the **lexer** will transform a string into a list of **tokens**.
  It is also the lexer that detects wrong characters.

- The **Parser**
  It calls the **Lexer** and applies the grammar in order to build the Abstract Syntax Tree (AST)
  It is also the parser that detects bad grammar.

- The evaluation
  The evaluation is the action of taking the AST built by the parser and returning the result of the operation.

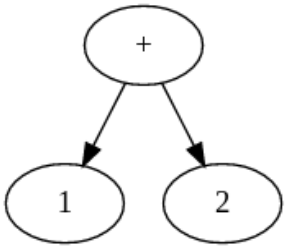
2.2.2 AST : what is it ?

An AST is a representation of an expression in a data structure that keeps the semantics.
For a mathematical expression, a binary tree is sufficient to keep the operations order.

Thus the operation :

```
> 1 + 2
```

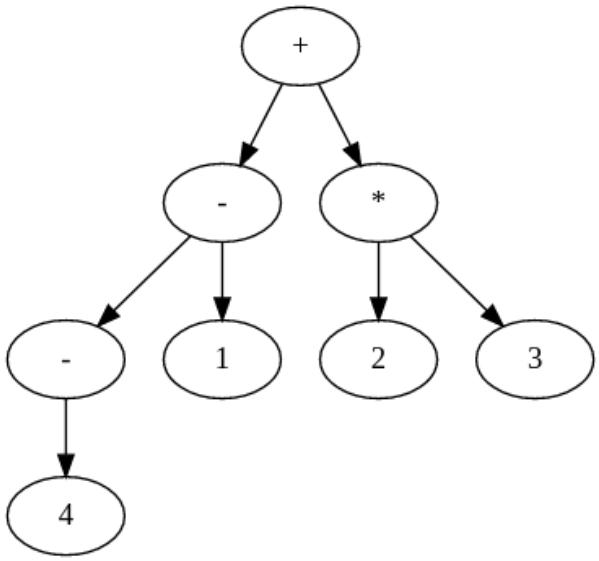
gives the following AST :



And the operation :

```
> -4 - 1 + 2 * 3
```

gives the following AST :



Thus, all internal nodes are operators and leaves are operands.
As you can see, an infix traversal rebuilds the given operation.

2.2.3 Mathematical expression : Definition

We can see that you still like mathematics, thus, in this tp, we will parse mathematical expressions. Mathematics being born before computers, expressions are absolutely not suitable to a machine, indeed :

- Operators do not have the same priority (the '*' and the '/' have to be evaluated before '+' and '-')
- Operators '+' and '-' have multiple meanings, they can be either binary or unary
- And because it is too easy, parentheses are here to add some difficulty

At the end of this tp, you should be able to parse and evaluate expressions like these :

```
> 42
> +42
> 31 + 11
> 2 + 10 * 4
> (2 + 1) * 11 + 9
> -3 + 54 - 9
> -  -  - -+(+-(-  -+(   ((    56+87) / 5) + 8) - -4) + 5 )* -1 + ++--7
```

All are equal to 42.
In order to define a mathematical expression, which is merely a sequence of characters, we will use a notion of language theory (this will come later in your studies) : the grammar of an expression.

A grammar looks like this :

```
expr:   '(' expr ')'
    |   ('+' | '-') expr
```

```
    |   expr ('+' | '-' | '*' | '/') expr
    |   INT

INT: [0-9]+
```

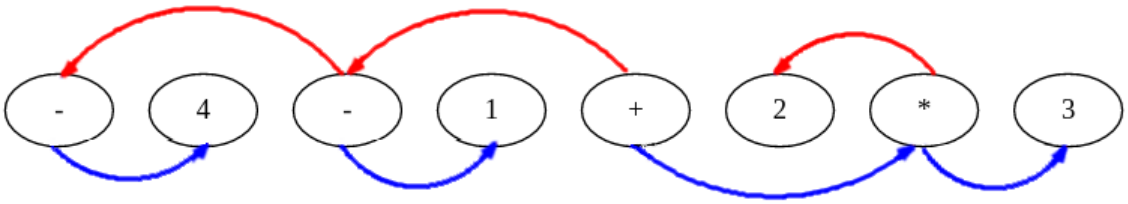
What you should understand :

- A mathematical expression can be :
  - An expression between parentheses
  - A unary '+' or '-' before an expression
  - Two expression separated by a binary operator
  - An integer
- An integer is a sequence of one or more digits.

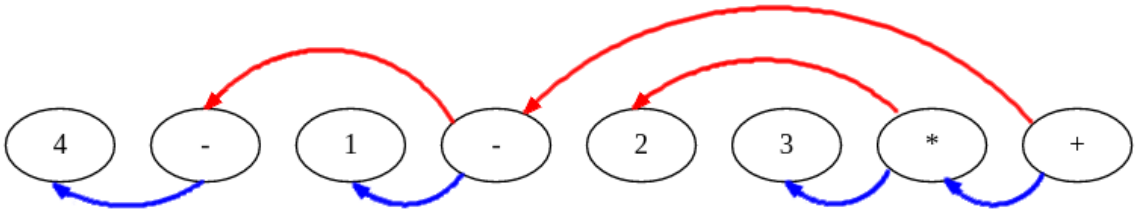
To simplify the construction of our AST, the parser will transform the expression in reverse Polish, thus, the expression will look like that :

```
> -4 - 1 + 2 * 3
4 - 1 - 2 3 * +
```

It seems impractical for us humans, but for a computer, this notation is much clearer. Indeed, if we look at the construction of the AST from the classic notation :
Blue arrows represent left relations and the red ones right relations.



While in reverse Polish :



As you can see, in reverse Polish, the children are always at the left of their parents. The construction of the AST is much simpler.

3 Exercices

3.1 Lists and generics

Warning!

For this part, using containers that already exist, like vectors, lists, ... is forbidden

In this part we will see how to write containers thanks to the generics. We will be able to use our containers to store any type without having to implement a new version each time that we want to add a new type in our container.

3.2 The structure

For the container that we are going to make, we will need to link our objects. It's a strategy where each element contains a link to the next one (and in our case the previous one). First thing, let's create a class that stores an object, a link to the next element and the previous one. The constructor has to initialize the links to *null* and store the object given as parameter:

```
1  public Node(T elt);
```

The only missing thing to our skeleton is an ensemble of *getter* and *setter* for the different attributes:

```
1  public T Data
2  {
3      get {}
4      set {}
5  }
6  public Node<T> Next
7  {
8      get {}
9      set {}
10 }
11 public Node<T> Prev
12 {
13     get {}
14     set {}
15 }
```

3.2.1 The list

We have created objects that are capable of storing data from different types and links between them, but they are pretty much useless for now. Let's fix it by coding a linked list.
The goal of this class will be to contain the algorithms for the construction, the destruction, the search and the insertion in our list.
The constructor must comply to this prototype:

```
1  public List(); //create an empty list
2  public List(T data); //create a list that contains only element.
```

The next thing to do is to write a function that prints all the elements in the list.
The elements will be separated by a comma and a space, the printing will end with a new line. Be careful not to put any comma at the beginning or at the end. This function will allow you to test your code in the following exercise:

```
1 public void print();
2 // elt1, elt2, elt3\n
```

It would be useful to be able to access the elements of our list by using the brackets like we do with the arrays, so we are going to implement this feature. To do so, we need to overload the [] operator, which works like a *getter/setter*.
This operator must allow the user to get or set a value on an existing index in the list. In the case of an index that doesn't exist, a *InvalidDataException* is expected. Here is the prototype:

```
1 public T this[int i]
2 {
3     get
4     {}
5     set
6     {}
7 }
```

To finish our container, we have to implement the insertion and the deletion at a given index. The prototypes are:

```
1 public void insert(int i, T value); //insert an element "value" at the index i;
2 public void delete(int i); //delete the value at the index i
```

Careful

Inserting an element at the end of the list must work, but beyond your program must stop with an *InvalidDataException*

Warning!

After a deletion or an insertion, the list must stay correctly chained.

That's it, your first container is ready, a bit rudimentary, but 100% usable.

3.2.2 The stack

The goal of this part is to create a class *stack* that inherits from our list, so it keeps its characteristics like the use of generics. This class adds the possibility of adding and deleting at the head of the list easily. This will follow the behavior of a usual stack. Let's begin:

```
1 public T front(); //return the element at the head of the stack
2 public void popFront(); //delete the element at the head
3 public void pushFront(T elt); //insert an element at the head
```

Remember to add the heritage for this class and the following ones.

This exercise may be pointless for you, but the use of these functions allows a small gain in performance if they are correctly implemented. This can be useful on a big number of calls.

3.2.3 The queues

In the same idea as the previous one, we are going to implement a *queue*. As you can imagine this means adding an element at the queue and deleting one at the head. You have to follow these prototypes:

```
1 public T front(); //return the element at the head
2 public void popFront();//delete an element at the head
3 public void pushBack(T elt);//insert an element at the queue
```

3.2.4 The deque (or double ended queue)

You may not know this container. This one allows us to delete and insert at both ends. The prototypes are as follows:

```
1 public T front(); //return the element at the head
2 public T back(); //return the element at the queue
3 public virtual void popBack(); //delete an element at the queue
4 public void popFront(); //delete an element at the head
5 public void pushFront(T elt); //insert an element at the head
6 public void pushBack(T elt); ////insert an element at the queue
```

That concludes this part.

3.3 Let's parse !

Warning!

"Premature optimization is the root of all evil" - Donald Knuth
Parsing is a complex notion, do not make the mistake of doing everything from the beginning, and work step by step.
However, do not forget that you will have to add new features throughout the project so keep your code clean and make comments !

Allowed functions

In this part you can use all containers in the C# library and the function *Int.Parse()*.
For all other functions, ask your ACDC.

We will start with simple mathematical expressions, i.e, with only binary operators and operands.

All expression will be valid expressions, we will not test with malformed one.

3.3.1 Lexing Luthoring

Before the lexing, you need to create the class *Token*, as we said previously, which will contain an *enum Type* and a *string*.
Write the two *getters* for those as well.
Write the constructor :

```
1 public Token(Type toktype, string val)
```

The following function is given :

```
1  public override string ToString()
```

It returns the string contained in the *token* and allows us to use I/O functions on a *token*. This function is used in the correction, so do not modify it.

Then, in *Lexer.cs*, write the function :

```
1  public static Token Tokenize(ref int pos, string expr)
```

Which looks at the character at the position *pos* in the *string expr* and returns the corresponding *token*.
The position must be updated to correspond to the character following the token :

```
1  int pos = 0;
2  Token tok = Lexer.Tokenize(ref pos, "542   * 6");
3  Console.WriteLine("pos : {0}, token : {1}", pos, tok);
4  // pos : 3, token : 542
```

Finally, write the function :

```
1  public static List<Token> Lex(string expr)
```

that takes a *string expr* as parameter and returns the corresponding list of **tokens**.

At the end of this part, the following code :

```
1  List<Token> tokens = Lex("   3   + 4 -5* 6");
2  foreach (Token token in tokens)
3      Console.Write(token);
4  Console.Writeline();
```

Should print this :

```
3+4-5*6
```

3.3.2 Some gardening

For our AST, we will use three different types of node :

- Binary operators
- Unary operators
- Operands

For now, we will forget unary operators.

As our nodes have different attributes but the same method, we will use an Interface (see TPCS9) *INode.cs*.

For now, it will have the following methods :

```
1  void Build(Stack<INode> output);
2  void Print();
```

The method *Build()* is simple. It uses a stack where elements are stacked in reverse Polish (the head is the element at the right).

- for a binary operator :
  - Dequeue and put the element as right son
  - Call *Build()* on the right son
  - Repeat for left son
- for an operand :
  - Do nothing

The method *Print()* must print the AST in infix order with parenthesis :

- For a binary operator :
  - Print a '('
  - Call *Print()* on the right son
  - Print the operator
  - Call *Print()* on the left son
  - Print a ')'
- For an operand :
  - Print the number

Thus :

```
Normal
> 4 - 1 + 2 * 3
Print()
> ((4-1)+(2*3))
```

important

The function *Print()* will be used for the correction of your AST. Be sure to follow the given format.

3.3.3 Parse me if you can

Now that you are able to transform an expression represented by a string into *tokens* and to build an AST, it is time to start the TP : it is time to parse.
You will have to write the following function :

```
1  public static INode Parse(string expr)
```

It will take a *string* to parse, call the Lexer, and return the root of the AST.

To do this we advise you to use the Shunting Yard algorithm [1] by Edsger Dijkstra (him again).
As you have noticed, the method *Build()* of our AST takes a stack so the output of your algorithm should be one.

[1] https://en.wikipedia.org/wiki/Algorithme_Shunting-yard

3.3.4 Some parkour

Now that our AST have grown, we need to browse it. In order to do so, add the function :

```
1  int Eval();
```

in the interface INode and in the class inheriting from it.

- For binary operators : return the computing of the operation carried by the node between the evaluation of the left and right children.

- For an operand : Return the value carried by the node.

3.3.5 Unary exploration

Now that you are capable of parsing and evaluating simple expressions, and handling operators priorities, it is time to work on the second difficulty of mathematical expressions : unary operators.
You will have to complete the class in the file *UnaryNode.cs* that contains a boolean *positive* and one child *val*.

Tip

During the parsing, we advise you to use a different stack to stock unary operators and dequeue it when necessary.

3.4 Bonus

3.4.1 hsiloP emoS

Implement the function :

```
1  public void PrintRevertPolish();
```

That must print the AST in reverse Polish notation :

```
Normal
> 4 - 1 + 2 * 3
Revert Polish
> 4 1 - 2 3 * +
```

It must have a trailing whitespace at the end to simplify the algorithms.

3.4.2 Someone said lisp ?

Now that you can parse every operator, it is time to use parentheses !
These must not be in the AST, only operators and operands can be here. However, the construction of your tree is altered with parentheses.

These violent deadlines have violent ends.