TP C#12 : One parser to rule them all

Consignes de rendu

A la fin de ce TP, vous devrez rendre une archive respectant l'architecture suivante :

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

N'oubliez pas de vérifier les points suivants avant de rendre :

- Remplacez `prenom.nom` par votre propre login et n'oubliez pas le fichier `AUTHORS`.
- Les fichiers `AUTHORS` et `README` sont obligatoires.
- Pas de dossiers `bin` ou `obj` dans le projet.
- Respectez scrupuleusement les prototypes demandés.
- Retirez tous les tests de votre code.
- **Le code doit compiler !**

AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (*), un espace, votre login et un retour à la ligne. Voici un exemple (où `$` est un retour à la ligne et ␣ un espace) :

```
*␣prenom.nom$
```

Notez que le nom du fichier est `AUTHORS` sans extension. Pour créer simplement un fichier `AUTHORS` valide, vous pouvez taper la commande suivante dans un terminal :

```
echo "* prenom.nom" > AUTHORS
```

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

1 Introduction

1.1 Objectifs

Dans ce TP vous allez voir deux notions importantes de programmation.
Les **génériques**, pouvoir coder un projet c'est bien, mais coder un projet qui permet de coder plusieurs autres projets c'est mieux. Vous allez pouvoir vous plonger dans le monde magnifique de la programmation générique !
Un parser : vous avez pu toucher à des évaluations simples, vous en avez bavé sur Mathmageddon et TinyBistro, il est temps de tout mélanger afin de savoir comment une expression mathématique est évaluée !

important

Les deux parties de ce tp sont complètement séparées, il n'est pas nécessaire de compléter l'une pour faire l'autre.

Pour toute information sur les différentes structures de données de ce tp : `https://ionisx.com`

2 Cours

2.1 Les génériques

Les **génériques** sont une notion de C# (et de certains autres langages) très utile et puissante. Cela permet de créer des fonctions et des classes qui peuvent accepter n'importe quel type (du moment que les opérations auquel les objets "génériques" sont soumis acceptent le dis type). Cette fonctionnalité permet de factoriser le code lorsque l'on veut appliquer le même comportement à plusieurs types différents.

Un exemple sera plus parlant :

```
1  //<T> précise que la fonction est générique, T est donc un type.
2  void print_line<T>(T elt)
3  {
4      Console.WriteLine(elt);
5  }
6  //On l'utilise de la manière suivante:
7  print_line<int>(3);
8  //Le T du générique est remplacé par int, la fonction sait donc quoi faire.
9  //Pour utiliser notre fonction avec une string il suffit de faire:
10 print_line<string>("These violent deadlines have violent ends.");
```

Si vous cherchez plus de précisions, nous vous invitons à lire cette page de la msdn qui y est dédiée : `https://msdn.microsoft.com/en-us/library/ms379564(v=vs.80).aspx`

2.2 Parsons

Le **parsing** est une notion tout aussi importante que sous-évaluée en informatique, vous y avez déjà goûté précédemment avec une évaluation simple comme le Brainfuck et le partage de données avec le JSON, nous allons ici vous montrer que ce n'est pas aussi simple que ce que vous avez vu.

En Brainfuck, vous avez du interpréter un langage basique qui se lit de gauche à droite et où chaque symbole n'avait qu'une seule signification (nous ne vous ferons pas ici l'affront de vous les rappeler, experts que vous êtes devenus). La question étant : comment fait-on pour les expressions qui ne s'évaluent pas de gauche à droite et dont le même symbole peut se répéter plusieurs fois ?

2.2.1 Mais un parser, comment ça fonctionne ?

Attention

Beaucoup d'anglicismes seront utilisés lors de ce tp, ceux-ci sont des conventions et sont moins longs à prononcer que leurs équivalents français

Un **parser** se divise en trois grandes catégories :

— L'analyseur lexical (ou **lexer**)
Celui-ci s'occupe de transformer les différents termes lexicaux en données facilement manipulables pour nous, que nous appelerons **tokens**.
Un **token** sera représenté par une classe qui contient :
— Une *enum* correspondant à l'entité lexicale
— Une *string* qui contient la partie qui correspond l'entité lexicale dans l'expression à parser
Ainsi le **lexer** s'occupera de transformer une chaîne de caractères en liste de **tokens**.
C'est aussi lui qui s'occupe de détecter les caractères anormaux.

— L'analyseur syntaxique (ou **parser**)
Il appelle le **Lexer** et s'occupe d'appliquer la grammaire afin de construire et renvoyer l'Arbre Syntaxique Abstrait (Abstract Syntax Tree que nous écourterons en AST).
C'est lui qui s'occupe de détecter les erreurs de syntaxe.

— L'évaluation (celui là ça va)
L'évaluateur récupère l'AST construit par le parser et retourne le résultat de l'opération.
Arrivé à cette étape, il n'y a plus d'erreurs à renvoyer.

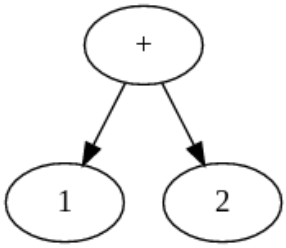
2.2.2 Mais un AST qu'est-ce que c'est ?

Un AST est une manière de représenter l'expression à évaluer dans une structure de donnée permettant de garder sa sémantique.
Pour une expression mathématique, une variante de l'arbre binaire est suffisante afin de représenter l'ordre des opérations.

Ainsi l'opération :

```
> 1 + 2
```

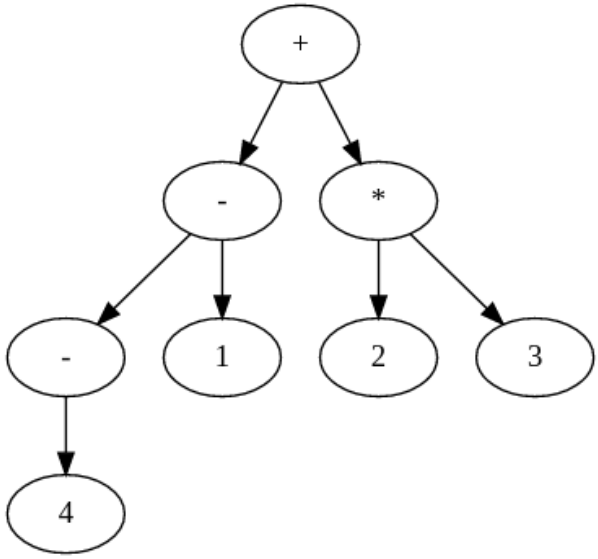
Donne l'AST suivant :



Et l'opération :

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

Donne l'AST suivant :



Ainsi tous les noeuds internes sont des opérateurs et toutes les feuilles sont des opérandes. Comme vous pouvez le remarquer, un parcours infixe de l'arbre permet de reconstruire l'opération et ainsi d'avoir le résultat.

2.2.3 Mais qu'est-ce qu'une expression mathématique ?

Nous sentons que vous aimez encore les maths, aussi, nous allons, dans ce tp, prendre le cas des expressions mathématiques. Les mathématiques étant arrivées avant l'informatique (si si, promis), la formation des expressions n'est absolument pas adaptée à une machine, en effet :

- Les opérateurs n'ont pas tous la même priorité (le '*' et le '/' doivent être traités avant le '+' et le '-').
- Les opérateurs '+' et '-' ont plusieurs significations ils peuvent être binaires ou unaires.
- Pour couronner le tout, les parenthèses sont là pour rajouter de la difficulté.

Ainsi à la fin de ce TP vous devez être capables de parser et d'évaluer des expressions de ce type :

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

Toutes ces opérations valent 42.

Pour pouvoir définir au mieux une expression mathématique qui n'est en soit qu'une suite de caractères, nous allons utiliser une notion de théorie des langages (qui arrivera plus tard dans vos études) : la grammaire d'une expression.

Une grammaire se présente de cette façon :

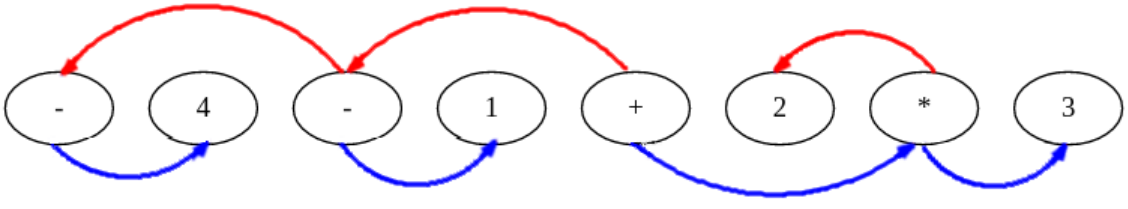
```
expr:   '(' expr ')'
    |   ('+' | '-') expr
    |   expr ('+' | '-' | '*' | '/') expr
    |   INT

INT: [0-9]+
```

Ce qu'il faut en déduire :
- Une expression mathématique peut être :
  - Une expression entre parenthèse
  - Un '+' ou un '-' unaire devant une expression
  - Deux expressions séparées par un opérateur binaire
  - Un entier
- Un entier est une composition d'au moins un chiffre entre '0' et '9' sans espaces

Pour simplifier grandement la construction de notre AST, le parser s'occupera de tranformer l'expression mathématique en polonaise inversée C'est à dire que l'expression précédente donnera :

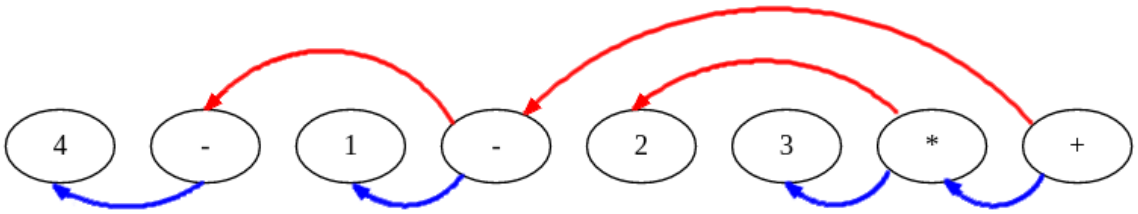
```
> -4 - 1 + 2 * 3
4 - 1 - 2 3 * +
```

Cela peut paraître peu pratique pour nous, humains, mais pour un ordinateur cette notation est beaucoup plus claire. En effet, si on regarde la construction de l'AST à partir de la notation normale :
Les flèches bleues représentant les relations gauches et les rouges les relations droites.



Tandis

qu'en polonaise inversée cela donne :



Comme vous pouvez le voir, en notation polonaise inversée tous les fils sont à gauche de leur parent tandis qu'en notation normale tout est inversé selon que le fils soit droit ou gauche.
La construction de notre AST s'en trouve simplifiée.

3 Exercices

3.1 Des listes et des génériques

Attention !

Pour cette partie il est interdit d'utiliser les conteneur préexistant tel que list, vector,...

Dans cette partie nous allons voir comment écrire des conteneurs grâce à des génériques. On pourra donc utiliser nos conteneurs pour stocker n'importe quel type sans avoir à réimplémenter une nouvelle version à chaque fois que l'on veut que le dit conteneur accepte un nouveau type.

3.1.1 La structure

Pour les conteneurs que nous allons réaliser nous aurons besoin de chainer nos objets. Il s'agit d'une stratégie où chaque élément contient un lien vers le suivant (et dans notre cas le précédent). Commençons par créer une classe qui stocke un objet ainsi qu'une référence vers l'objet suivant et le précedent.
Pour le constructeur il suffit d'initialiser les liens à *null* et de stocker l'objet passé en paramètre :

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

La seul chose qu'il manque à notre squelette est un ensemble de *getter* et *setter* pour ses différents attributs :

```
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 }
```

3.1.2 La liste

Nous avons donc créé des objets capables de stocker des donnéss de différents types et de se lier entre eux, mais nous n'en faisons pas grand chose pour le moment. Nous allons remédier à ce problème en codant notre premier conteneur, une liste chainée.
Le but de cette classe sera de contenir des algorithmes permattant la construction, la destruction mais aussi la recherche et l'insertion dans la chaine que nous avons créée plus tôt.
Commençons par le début, les constructeurs :

```
1  public List(); //créer une liste vide
2  public List(T data); //créer une liste ne contenant qu'un seul élément
```

Poursuivons en écrivant une fonction qui affiche tous les éléments contenus dans la liste.
Les éléments seront séparés par une virgule et un espace, l'affichage se terminera par un retour à la ligne. Attention de ne pas mettre de virgule au début ni à la fin. Cette fonction vous permettra par la suite de tester ce que vous faites :

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

Ce serait pratique de pouvoir accéder aux éléments de notre liste en utilisant les crochets comme avec un simple tableau, non ? Eh bien nous allons faire en sorte que cela soit possible.
Pour cela il suffit de surcharger l'opérateur [ ], cela fonctionne comme un *getter/setter*. Cette opérateur doit permettre de récupérer ou remplacer un élément à un index existant dans la liste. Dans le cas ou l'index entre crochet n'est pas dans la liste une *InvalidDataException* est attendue. A vous de jouer :

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

Pour finir notre conteneur, il ne nous manque plus que l'insertion et la suppression à un index donné. Voici les 2 prototypes :

```
1 public void insert(int i, T value); //insert l'élément "value" à l'index i;
2 public void delete(int i); //suprime la valeur à l'index i
```

Attention

Insérer un élément à la fin de la liste doit fonctionner, mais au dela votre programme doit s'arrêter avec une *InvalidDataException*

Important !

Après une suppression ou une insertion, la liste doit rester correctement chainée.

Voilà votre premier conteneur est opérationel, un peut rudimentaire, soit mais 100% utilisable.

3.1.3 Les piles

Nous allons créer une classe *stack* héritant de notre liste, elle gardera donc les caractéristiques de notre liste comme l'utilisation des génériques. Cette nouvelle classe ajoute à ce comportement la possibilité d'ajouter et supprimer un élément en tête de liste facilement. Cela permet d'imiter le comportement d'une pile classique. Commençons :

```
1 public T front(); //renvoie l'élément en tête de pile
2 public void popFront(); //supprime un élément en tête
3 public void pushFront(T elt); //insert un élément en tête
```

Pensez à ajouter l'héritage pour cette classe mais aussi pour les suivantes.

Cette exercice ne vous paraît peut être pas très utile, mais utiliser ces fonctions permettent un petit gain en performance si elles sont implémentées correctement, surtout si ces fonctions sont appelées répétitivement un grand nombre de fois.

3.1.4 Les files

Dans le même esprit que pour la *pile* nous allons implémenter une *file*. Nous pourrons donc ajouter des éléments en queue et en supprimer en tête. Voici les prototypes à suivre :

```
1 public T front(); //renvoie l'élément en tête
2 public void popFront();//supprime un élément en tête
3 public void pushBack(T elt);//insert un élément en queue
```

3.1.5 Les deque (ou file à double entrées)

Voilà un conteneur que vous ne connaissiez peur-être pas. Ce dernier permet d'ajouter et supprimer en tête et en queue. Voici qui promet pas mal de possibilités. Les prototypes à suivre sont les suivants :

```
1 public T front(); //renvoie l'élément en tête de deque
2 public T back(); //renvoie l'élément en queue de deque
3 public virtual void popBack(); //suprime un élément en queue
4 public void popFront(); //suprime un élément en tête
5 public void pushFront(T elt); //insert un élément en tête
6 public void pushBack(T elt); ////insert un élément en queue
```

Voici qui conclut cette partie sur les conteneurs chainés.

3.2 Salut ça parse ?

Attention !

"Premature optimization is the root of all evil" - Donald Knuth
Un parser est une notion difficile à mettre en place, ne faites pas l'erreur de vouloir gérer tous les cas dès le début et allez-y étape par étape.
Toutefois n'oubliez pas que pour ajouter de nouvelles fonctionnalités vous devrez revenir sur votre code donc prenez votre temps et commentez !

Fonction autorisées

Dans cette partie vous avez le droit d'utiliser les conteneurs présents dans la bibliothèque C# et la fonction *Int.Parse()*.
Pour toute autre fonction, demandez à vos ACDC.

Nous commencerons par des expressions mathématiques simples, c'est à dire avec uniquement les opérateurs binaires et les opérandes.

Toutes les expressions seront valides, nous ne testerons pas avec des malformées.

3.2.1 Lexons Extendons

Avant d'attaquer l'analyse lexicale, il faut créer la classe *Token*, comme dit plus haut celle-ci aura comme attribut une *enum Type* et une *string*.
Implémenter aussi les deux *getter* des attributs.
Coder le constructeur :

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

La fonction suivante vous est donnée :

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

Elle retourne la string contenue dans le *token* et permet d'utiliser les fonctions I/O sur un *token*, Celle-ci servant à la correction, ne la modifiez surtout pas.

Puis dans *Lexer.cs*, coder la fonction :

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

Qui regarde le caractère à la position *pos* dans la chaine de caractères *expr* et retourne le *token* correspondant.
Enfin, coder la fonction :

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

Qui prend une chaine de caractères *expr* en paramètre et retourne la liste de **tokens** correspondante.

À la fin de cette partie, le morceau de code suivant :

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

Devrait afficher ceci :

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

3.2.2 Un peu de jardinerie

Pour notre AST, nous allons créer trois types de noeuds différents :
— Les opérateurs binaires
— Les opérateurs unaires
— Les opérandes
Pour l'instant nous allons oublier les opérateurs unaires.

Nos types de noeuds ayant des attributs différents mais devant avoir les mêmes méthodes nous allons utiliser une Interface (voir TPCS9) *INode.cs*.

Pour l'instant, celle-ci ne contient que les méthodes suivantes :

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

La méthode *Build()* est simple, elle utilise une pile dont les élément sont empilés en polonaise inversée (Le haut de la pile correspond à l'élément le plus à droite).

- Pour un opérateur binaire :
  - Dépiler et mettre l'élément en fils droit
  - Appeler *Build()* sur le fils droit
  - Faire de même pour le fils gauche
- Pour une opérande :
  - Ne rien faire

La méthode *Print()* doit imprimer l'AST de cette manière :

- Pour un opérateur binaire :
  - Écrire une '('
  - Appeler *Print()* sur le fils gauche
  - Écrire l'opérateur
  - Appeler *Print()* sur le fils droit
  - Écrire une ')'
- Pour une valeur :
  - Écrire le nombre

Ainsi :

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

important

La fonction *Print()* servira a la correction de votre AST, veillez à bien respecter le format demandé.

3.2.3 Parse qu'il faut bien le faire

Maintenant que vous êtes capables de tranformer une expression sous forme de chaine de caractères en *tokens* et de construire un AST, il est temps de passer aux choses serieuses : Le parser.
Il vous faudra coder la fonction :

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

Celle-ci prend en paramètre la chaine de caractère à parser, appelle le lexer et retourne la racine de notre AST.

Pour ce faire nous vous conseillons vivement le shunting yard algorithm [1] de Edsger Dijkstra (oui, encore lui).
Vous l'aurez deviné, la méthode *Build()* de notre AST prenant une pile, la sortie de votre algorithme devra en être une.

3.2.4 Un peu d'accrobranche

Maintenant que notre bel AST a poussé, il faut le parcourir, pour cela ajoutez la fonction :

1. https://fr.wikipedia.org/wiki/Algorithme_Shunting-yard

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

Dans l'interface INode et dans les classes en découlant.

- Pour un opérateur binaire : renvoyer le résultat de l'opération portée par le noeud entre l'évaluation du fils gauche et l'évaluation du fils droit.
- Pour une opérande : renvoyer la valeur portée par le noeud.

3.2.5 Exploration unaire

Maintenant que vous êtes capables de parser et évaluer des expressions simples, et de gérer les priorités opératoires, il est temps de passer à la deuxième difficulté des expressions mathématiques : les opérateurs unaires.
Il vous faudra compléter la classe présente dans *UnaryNode.cs* qui contient un simple booléen *positive* et un seul fils *val*.

Conseil

Pednant la phase de parsing, nous vous conseillons d'utiliser une pile à part pour stocker les opérateurs unaires et de la dépiler quand nécessaire.

3.3 Bonus

3.3.1 esianoloP ed uep nU

Implémenter la fonction :

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

Qui doit print l'AST en polonaise inversée :

```
Normale
> 4 - 1 + 2 * 3
Polonaise inversée
> 4 1 - 2 3 * +
```

Il doit y avoir un espace à la fin afin de faciliter les algorithmes.

3.3.2 Quelqu'un a dit lisp ?

Maintenant que vous savez parser tous types d'opérateurs il est temps de s'attaquer aux parenthèses !
Celles-ci ne doivent pas se retrouver dans l'AST, seuls les opérateurs et les opérandes y ont leurs places, toutefois la construction de celui-ci est différente avec les parenthèses.

These violent deadlines have violent ends.