TP C#8 : Brainfuck and co

Consignes de rendu

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

```
prenom.nom.zip
|-- prenom.nom/
    |-- AUTHORS
    |-- README
    |-- Brainfuck/
        |-- Brainfuck.sln
        |-- Brainfuck/
            |-- 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).

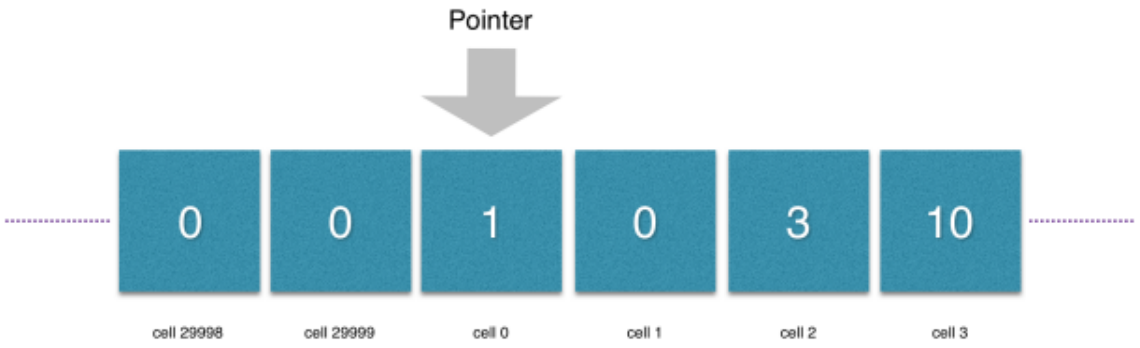
WARNING

Ce TP demande du temps. Si vous le commencez seulement quelques jours avant la deadline vous ne réussirez pas à avoir une bonne note.

1 Brainfuck

1.1 Généralités

Le Brainfuck est un langage de programmation exotique créé par Urban Müller en 1993. L'objectif de ce langage est d'être le plus simple possible mais en restant Turing-Complet. Il est donc théoriquement possible d'écrire n'importe quel programme en Brainfuck. Tout comme la machine de Turing le langage est composé d'un "ruban" ou d'un tableau stockant des octets. Il est possible de se déplacer d'une case à l'autre. Un pointeur permet de garder la position de la case courante du tableau.



Le tableau est généralement un tableau circulaire de 30000 cases.

1.2 Instructions

Le langage est composé de seulement 8 symboles différents, chaque symbole correspondant à une instruction. Voici la liste de ces instructions :

— > incrémente le pointeur p
— < décrémente le pointeur p
— + incrémente la valeur de la case du tableau pointée par le pointeur p
— **-** décrémente la valeur de la case du tableau pointée par le pointeur p
— **.** affiche le caractère pointé par le pointeur p(valeur case = code ASCII)
— **,** lit un caractère sur l'entrée standard et le stocke dans la case courante du tableau
— [saute à l'instruction après le] correspondant si l'octet pointé est à 0
—] retourne à l'instruction après le [si l'octet pointé est différent de 0

```
1 ++++++++++[>+++++++>++++++++++>+++>+<<<<-]
2 >++>+.+++++++..+++>++.<<+++++++++++++++.
3 >.+++.------.--------.>+.>.
```

Ce code affiche "**Hello World !**" suivit d'un **retour à la ligne**. La première ligne fait deux choses. Tout d'abord elle initialise un compteur à dix dans la première case du tableau. Une fois fait, elle lance une boucle afin d'initiaiser les quatre cases suivantes avec les valeurs les plus proches de celles des caractères à afficher. La seconde case est à 70 pour les lettres majuscules. La troisième à 100 pour les minuscules. La quatrième à 30 pour les caractères spéciaux et la cinquième a 10 pour la nouvelle ligne.
Le reste du code consiste juste à incrémenter/décrémenter les cases pour atteindre la valeur ascii du caractère à afficher.

1.3 implémentation

Vous l'avez compris l'objectif de cette première partie est de coder un interpréteur de Brainfuck. Le tableau sera un tableau d'**Int** de taille 30000. Le tableau devra être géré de façon circulaire. Le pointeur sera implémenté sous forme d'un Int qui représentera l'index de la case pointée. Les valeurs stockées devront être comprises entre **0** et **255** inclus. N'oubliez pas de gérer l'overflow (la valeur **256** passe à **0** et la valeur **-1** passe à **255**).

2 Dictionary

2.1 Définition

La classe **Dictionary** permet de gérer en ensemble de clef - valeur. C'est un outil puissant permettant l'ajout et l'accès d'une valeur via une clef associée en un temps constant. Cette efficacité est obtenue grâce à une implémentation sous forme de hash table. Pour plus d'informations sur l'implémentation ou le mode de comparaison des clés, lire la parties "Notes" de la page msdn : `https://msdn.microsoft.com/fr-fr/library/xfhwa508(v=vs.110).aspx`

2.2 Utilisation

Dans ce TP, il vous sera demandé d'utiliser cette classe afin d'implémenter un interpréteur brainfuck supportant un ensemble dynamique de symboles. Il ne vous sera demandé que les aspects les plus basiques de cette classe. Tout ce dont vous aurait besoin est montré dans

l'exemple qui suit. Si vous utilisez d'autres notions il y a de grandes chances que vous soyez sur la mauvaise voie.

```
1  using System;
2  using System.Collections.Generic;
3
4  public class Example
5  {
6      public static void Main()
7      {
8          // Create a new dictionary of strings, with string keys.
9          Dictionary<string, string> acdcs =
10             new Dictionary<string, string>();
11
12         // Add some elements to the dictionary. There are no
13         // duplicate keys, but some of the values are duplicates.
14         acdcs.Add("Zarakailloux", "The unknown");
15         acdcs.Add("Kirszie", "El Major");
16         acdcs.Add("Heldhy", "Cuttie.exe");
17
18         // The Add method throws an exception if the new key is
19         // already in the dictionary.
20         try
21         {
22             acdcs.Add("Heldhy", "NoOneWillNotice");
23         }
24         catch (ArgumentException)
25         {
26             Console.WriteLine("An acdc with Key = \"Heldhy\" already
27             exists.");
28         }
```

```
1           // The Item property is another name for the indexer, so you
2           // can omit its name when accessing elements.
3           Console.WriteLine("For key = \"Zarakailloux\", value = {0}.",
4               acdcs["Zarakailloux"]);
5
6           // The indexer can be used to change the value associated
7           // with a key.
8           acdcs["Zarakailloux"] = "JSON master";
9           Console.WriteLine("For key = \"Zarakailloux\", value = {0}.",
10              acdcs["Zarakailloux"]);
11
12          // If a key does not exist, setting the indexer for that key
13          // adds a new key/value pair.
14          acdcs["DotH"] = "Source files destroyer";
15
16          // The indexer throws an exception if the requested key is
17          // not in the dictionary.
18          try
19          {
20              Console.WriteLine("For key = \"Gollum\", value = {0}.",
21                  acdcs["Gollum"]);
22          }
23          catch (KeyNotFoundException)
24          {
25              Console.WriteLine("Key = \"Gollum\" is not found.");
26          }
27          // When a program often has to try keys that turn out not to
28          // be in the dictionary, TryGetValue can be a more efficient
29          // way to retrieve values.
30          string value = "";
31          if (acdcs.TryGetValue("Coucou", out value))
32              Console.WriteLine("For key = \"Coucou\", value = {0}.", value);
33          else
34              Console.WriteLine("Key = \"Coucou\" is not found.");
35
36          // ContainsKey can be used to test keys before inserting
37          // them.
38          if (!acdcs.ContainsKey("TheCoon"))
39          {
40              acdcs.Add("TheCoon", "Ninja");
41              Console.WriteLine("Value added for key = \"TheCoon\": {0}",
42                  acdcs["TheCoon"]);
43          }
```

```
        // When you use foreach to enumerate dictionary elements,
        // the elements are retrieved as KeyValuePair objects.
        Console.WriteLine();
        foreach (KeyValuePair<string, string> kvp in acdcs)
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }

        // Use the Remove method to remove a key/value pair.
        Console.WriteLine("\nRemove(\"TheCoon\")");
        acdcs.Remove("TheCoon");

        if (!acdcs.ContainsKey("TheCoon"))
            Console.WriteLine("Key \"TheCoon\" is not found.");
    }
}

/* This code example produces the following output:

An acdc with Key = "Heldhy" already exists.
For key = "Zarakailloux", value = The unknown.
For key = "Zarakailloux", value = JSON master.
Key = "Gollum" is not found.
Key = "Coucou" is not found.
Value added for key = "TheCoon": Ninja

Key = Zarakailloux, Value = JSON master
Key = Kirszie, Value = El Major
Key = Heldhy, Value = Cuttie.exe
Key = DotH, Value = Source files destroyer
Key = TheCoon, Value = Ninja

Remove("TheCoon")
Key "TheCoon" is not found.
 */
```

3 JSON

3.1 Introduction

Le JSON, ou JavaScript Object Notation est un format de données qui dérive du Javascript. Il permet de représenter des données et de les structurer comme le ferait XML ou YAML par exemple. L'avantage principal du JSON par rapport à ces autres formats est la légèreté. Il est en effet très facile de lire du JSON à l'oeil nu et de le comprendre, et il est à maitriser car ses règles sont simples.

Cependant, le JSON souffre de certaines limitations, comme par exemple l'absence de commentaires dans sa version officielle, ou le fait qu'il se limite à des types simples.

3.2 Fonctionnement

En JSON il existe 6 types de valeurs. Nous allons vous les présenter ici dans une version simplifiée :

- La chaine de caractères. Elle commence par une double-quote '"' et finit également par une double-quote. Si l'on souhaite mettre le caractère '"', il faut l'échapper avec un backslash '\'.
- Le nombre. Pour ce TP, nous considérerons qu'un nombre peut eventuellement commencer par le symbole '-', puis par une suite de chiffres. On ne considèrera que les nombres entiers.
- Le booléen. C'est soit "true", soit "false", en minuscule. La différence avec une *string* se fait par l'absence de double-quotes autour du mot-clé.
- La valeur nulle. C'est tout simplement le mot-clé "null", également sans double-quote.
- La liste. Les listes en JSON n'ont pas de type, c'est à dire que l'on peut avoir un nombre et une chaîne de caractères dans la même liste. Une liste commence par le symbole '[' et fini par un ']'. Les élements de la liste sont séparés par des virgules. Il ne doit pas y avoir de virgule à la fin de la liste, et on va considérer qu'une liste ne peut être vide.
- L'objet, ou le dictionnaire. C'est une extension de la liste, mais au lieu d'avoir une valeur simple il y a une association clé/valeur, comme les dictionnaires présentés dans la partie 2. La clé est forcément une chaine de caractères, mais comme la liste, la valeur peut être n'importe quel autre type JSON. Un objet commence par '{' et finit par '}'. Les éléments de la liste sont sous la forme "clé : valeur".

Un fichier JSON contient forcément un objet comme premier élément. C'est cet objet qui va contenir le reste.

Par exemple :

```
{
    "MEILLEURS_ACDC": 2020,
    "Campus": "Villejuif",
    "salles":
    [
        306,
        "Labo"
    ],
    {
        "TPC#" : true,
        "TPCAML": false,
        "Elèves qui ont eu 0": null
    }
}
```

L'indentation n'est pas prise en compte et n'est présente qu'à des fins de lisibilité. Ainsi le JSON suivant est également correct :

```
{"clé":"valeur",[1,2,3]}
```

Pour plus de détails, se référer au site `http://json.org`

4 Exercises

4.1 Brainfuck

Un interpréteur brainfuck classique est bien trop facile pour des jeunes talents tels que vous, et nous en sommes bien conscients. C'est la raison pour laquelle votre interpréteur va devoir supporter un ensemble **dynamique** de symboles. Pour ce faire, chaque fonction prend en paramètre une string et un objet *Dictionary<char, char>*. L'objet *Dictionary<char, char>* représente le couple <**symbol usuel**, **symbol exotique**>. Toutes les comparaisons et opérations impliquant un symbole devront être faites via le dictionnaire. Par exemple :

```
1 if (code[i] == symbols['>'])
2     do_something();
```

Les arguments sont passés aux fonctions via l'IDE. Pour le moment, si vous voulez jouer avec les symboles, il va falloir modifier la fonction *classicBrainfuck()* dans **Program.cs**.

4.1.1 I did not mean that

```
1 public static string Interpret(string code, Dictionary<char, char> symbols)
```

La méthode **Interpret** permet d'interpréter un code brainfuck donné en paramètre suivant l'ensemble de symbole dans **symbols**. Plusieurs étapes :

- Déclarations des variables selon les spécifications de l'implémentation définie plus haut.
- Initialisation du tableau
- Pour chaque caractère, réaliser l'instruction correspondante.

Si vous tombez sur un symbole qui n'appartient pas au dictionnaire (une erreur donc), vous devez déclenchez une exception de votre choix, tant qu'elle a du sens.
Protip : Pour la gestion des boucles vous pouvez utiliser une **pile** (cf MSDN et Cours d'algo) pour stocker leurs positions dans le code.

4.1.2 What do you mean

```
1 public static string GenerateCodeFromText(string text Dictionary<char, char> symbols)
```

Cette méthode renvoie le code Brainfuck permettant d'écrire la phrase contenue dans **text**. Bien entendu le code brainfuck doit suivre les symboles présents comme valeur dans le dictionnaire.

Protip : Le code "[-]" permet de remettre la case actuelle à 0.

4.1.3 The size does count

```
1 public static string ShortenCode(string program,
2                                   Dictionary<char, char> symbols)
```

ShortenCode doit renvoyer un code fonctionnel faisant la même chose que le code original mais avec moins de caractères. Vous pouvez utiliser toutes les techniques que vous voulez du moment que le code produit est plus court que le code original.
Pour la partie obligatoire de cet exercice vous avez à repérer les longues séquences de **symbols['+']** pour les remplacer par des multiplications (à l'aide de boucles). Si d'autres techniques sont utilisées, merci de le mentionner dans le Readme pour valider le Bonus.
Bien entendu, le code généré doit avoir exactement le même comportement.

4.2 JSON

Le but de ces exercices est de pouvoir lire un fichier JSON et créer des objets C# le représentant pour pouvoir le manipuler. Pour cela, nous allons coder un parser.
On considerera le fichier JSON comme valide.
L'utilisation de fonctions de C# pré-existante dans la bibliothèque de C# est strictement interdite et sera considérée comme de la triche.

4.2.1 JSONElement

Pour représenter un element JSON (parmi les 6 qui existent), nous allons utiliser une class **JSONElement** qui est fournie. Cela permettra d'avoir des *List<>* de *JSONElement* et de simplifier grandement votre code. On définit un enum *JSONType* qui contient les différents types d'éléments JSON et qui va nous permettre de nous y retrouver.
La classe *JSONElement* possède donc un type et plusieurs variables publiques. Il va falloir modifier uniquement la variable associée au type de notre objet *JSONElement*. Ainsi, si le type est 'NB', on ne va modifier que *int_value*.

4.2.2 GetJsonType

```
1  public static JSONElement.JSONType GetJsonType(char c)
```

Le but de cette fonction est de déterminer quel va être le type de l'élément en fonction du premier caractère (l'argument c), et de le renvoyer

4.2.3 ParseString

```
1  public static string ParseString(string json, ref int index)
```

Cette fonction prend en entrée une chaîne de caractères et l'index actuel au sein de cette chaîne par référence, et renvoie la chaîne de caractères lue. L'index donné pointera vers le '"' du début de la chaîne.

Par exemple :

```
1  int i = 2;
2  string input = "{ \"ACDC\": 2020}";
3  input[i] // '"'
4  ParseString(input, ref i); //renvoi "ACDC" et i vaut 8
```

4.2.4 ParseInt

```
1  public static int ParseInt(string json, ref int index)
```

Le même comportement que **ParseString** est attendu, mais cette fois-ci pour les *int*.

4.2.5 ParseBool

```
1  public static bool ParseBool(string json, ref int index)
```

Le même comportement que **ParseString** est attendu, mais cette fois-ci pour les *booléens*.

4.2.6 EatBlank

```
1 public static void EatBlank(string json, ref int index)
```

Le but de cette fonction est de faire avancer l'index passé en référence jusqu'à tomber sur le prochain caractère qui n'est pas "blanc" (Tout ce qui n'est pas un espace, une tabulation, ...)

4.2.7 ParseJSONString

```
1 public static JSONElement ParseJSONString(string json, ref int index)
```

La grosse fonction tant attendue qui parse un élément JSON de la chaîne de caractères. La chaine de caractères contient donc l'intégralité d'un fichier JSON que vous devez parser.
Pensez bien à utiliser les fonctions codées précedemment.

4.2.8 ParseJSONFile

```
1 public static JSONElement ParseJSONFile(string file)
```

Cette fonction prends un path vers un fichier JSON valide en entrée, et renvoie un **JSONElement** correspondant au fichier que vous venez de parcourir.

4.2.9 PrintJSON

```
1 public static void PrintJSON(JSONElement el)
```

Cette fonction affiche le contenu de l'objet *el*. Vous devez afficher un code JSON correct, cependant vous n'êtes pas obligés de respecter les espaces et tabulations (voir bonus).

4.2.10 SearchJSON

```
1 public static JSONElement SearchJSON(JSONElement element, string key)
```

Recherche une valeur d'un élément par sa clé. L'élément peut se trouver dans un dictionnaire lui même dans un dictionnaire, etc. Si la valeur n'existe pas dans l'objet JSON, vous devez renvoyer **null** (la valeur nulle de *C#*).

4.2.11 Brainfuck with options - BONUS

Maintenant que vous avez **SearchJSON**, vous pouvez charger un JSON contenant les 8 symboles du brainfuck classique comme clé et des valeurs associées. Le bouton **load brainfuck** permettra de changer l'implémentation courante vers celle représentée par le JSON.

4.2.12 Pretty PrintJSON - BONUS

Modifiez votre fonction PrintJSON pour que le **JSONprint** respecte la mise en forme traditionnelle du JSON (se référer au site `http://jsonprettyprint.com`)
Attention ! Vous ne devez pas modifier le prototype de la fonction, mais vous pouvez cependant l'utiliser comme fonction chapeau.

4.2.13 ParserJSON - BONUS

Comme bonus, vous pouvez implémenter un parser JSON qui gère les erreurs et est capable donc de détecter un JSON mal écrit. Vous devez déclencher une exception, celle que vous voulez tant qu'elle a du sens, si vous détectez un JSON syntaxiquement incorrect.

4.2.14 Ook ! - BONUS

Adaptez votre code pour qu'avec le bon fichier JSON de configuration, vous puissiez lire le **Ook !** (`http://www.dangermouse.net/esoteric/ook.html`).

These violent deadlines have violent ends.