TP C#7 : You and Cthulhu

Consignes de rendu

A la fin de ce TP, vous devez respecter l'architecture suivante :

```
csharp-tp7-prenom.nom/
|-- README
|-- YouAndCthulhu/
    |-- YouAndCthulhu.sln
    |-- YouAndCthulhu/
        |-- Tout sauf bin/ et obj/
```

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

— Le code **doit** compiler !
— Le fichier `README` est obligatoire.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.

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 bonus que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

1 Introduction

Marre de faire des travaux pratiques ? Vous avez l'impression que ce que vous programmez ici est inutile, et qu'un projet de jeu vidéo Unity est bien plus amusant ? Dommage, car cette semaine, vous allez ~~faire de l'ASCII art~~ travailler sur un petit interpréteur pour un langage que personne n'utilise ! Dans le cours de ce TP, nous parlerons de plusieurs sujet :

La première section sera à propos des exceptions, et comment les utiliser. Nous les avons déjà mentionnées dans le TP précédent, et nous allons donc approfondir un peu le sujet.

La seconde section vous aidera à comprendre ce que sont les langages ésotériques, et plus particulièrement celui que vous implémenterez cette semaine : **Cthulhu**.

Enfin, la dernière section se concentrera sur une notion particulière qui sera utile dans la conception de cet interpréteur : les expressions lambdas, et les Actions.

Lisez le cours dans son intégralité avant de passer à la partie exercice !

1.1 Note

Ce TP requiert l'utilisation de plusieurs objets C#, comme les Lists. N'oubliez pas de jeter un oeil à la MSDN pour vous informer.

Une tarball sera disponible sur l'intranet. Elle contient une architecture de projet complète avec des fonctions à implémenter, ainsi que quelques fonctions Cthulhu sous forme de fichiers test et un interpreteur Cthulhu exécutable avec lequel vous pourrez comparer les résultats du vôtre.

2 Cours

2.1 Exceptions

2.1.1 Kesako ?

Des exceptions sont lancées lorsqu'une instruction ne peut pas être exécutée. Vous avez probablement déjà rencontré certaines de ces situations :

```
1 int[] myArray = new int[10];
2 int a = myArray[10]; // Hors limite puisque les tableaux sont indexés à 0
```

```
1 int result = 42/0; // Division par 0
```

2.1.2 Comment les gérer ?

C# possède une syntaxe particulière pour gérer ces exceptions. Ils peuvent être interceptés, vous pouvez par exemple intercepter une erreur, générer un message personnalisé pour savoir d'où provient l'erreur et renvoyer l'erreur pour arrêter votre programme. Ainsi, la gestion des exceptions C # repose sur quatre mots clés : *try*, *catch*, *finally*, et *throw*.

- **try** : Ce bloc de code contient normalement le code considéré comme "dangereux". Il est suivi d'un ou plusieurs blocs catch.
- **catch** : Le programme intercepte l'exception avec un gestionnaire d'exceptions, qui exécute certaines instructions que vous souhaitez effectuer dans le cas où vous rencontriez cette exception.
- **finally** : Ce bloc est toujours exécuté, qu'une exception soit levée ou non. Faites attention à ce que vous mettez dans ce bloc, qui n'est pas obligatoire pour la syntaxe C #. N'oubliez pas que le code ici sera exécuté dans les deux cas. Par exemple, lorsque vous ouvrez un fichier, à la fin, vous devez le fermer dans tous les cas. (Ceci est une excellente utilisation de `finally` !).
- **throw** : En utilisant `throw`, vous pouvez faire en sorte que le programme lance une exception lorsqu'un problème se présente. Cette commande arrêtera le programme en cours d'exécution.

Attention

Chaque fois que vous détectez une erreur, vous devez toujours lever une exception. Sinon, le programme peut planter ou pire, il donnera l'impression que tout va bien et votre résultats seront corrompus.

2.1.3 Différentes classes d'exceptions

En C#, les exceptions sont représentées par des classes qui sont principalement dérivées directement ou indirectement de la classe `System.Exception`. Certaines exceptions standard sont bien connues et ce sont celles que vous rencontrerez la plupart du temps. Par exemple, `ArgumentException` ou `ArgumentOutOfRangeException`. Vous pouvez trouver toutes ces exceptions sur le MSDN, en cliquant sur ce lien.

2.1.4 Exemple

```
1  public static int Main(string[] args)
2  {
3      int res = 0;
4      Console.Write("> ");
5      string read = Console.ReadLine();
6      try
7      {
8          while(read != "")
9          {
10             res += Int32.Parse(read);
11             Console.WriteLine(res);
12             Console.Write("> ");
13             read = Console.ReadLine();
14         }
15     }
16     catch(FormatException e) // L'entrée de l'utilisateur n'est pas un nombre
17     {
18         Console.WriteLine("The user did not input a number!");
19         throw e; // Throw back the exception
20     }
21     catch(Exception e) // Autre exception
22     {
23         Console.WriteLine("Oops");
24         throw e;
25     }
26     finally
27     {
28         Console.WriteLine("This block is always executed!");
29     }
30     return res;
31 }
```

2.2 Langage ésotérique

Un *langage ésotérique* (dit aussi *esolang*) est un langage de programmation conçu pour tester les limites de la conception d'un langage de programmation informatique, comme preuve de concept, art logiciel, interface de piratage vers un autre langage (en particulier les langages de programmation fonctionnelle ou procédurale), ou tout simplement comme blague.

Fun fact !

+[-[«[+[—>]-[«<]]]»>-]>-.—.>..>.««-.<+.»»>.>.«.<-.
signifie "Hello World !" en Brainfuck, un langage ésotérique.

Habituellement, un esolang est créé sans intention d'être utilisé pour la programmation "grand public", bien que certaines fonctionnalités ésotériques, telles que la syntaxe *visuospatiale*, aient inspiré des applications pratiques dans les arts. Ses créateurs savent que sa conception rendra très probablement le code difficile et illisible, il est donc inutilisable pour écrire des logiciels et développer des applications.

Ces langues sont souvent populaires parmi les pirates et les amateurs. Leur objectif habituel est de supprimer ou de remplacer les fonctionnalités de langage conventionnelles tout en conservant un langage Turing-complet, ou même pour lequel la classe de calcul est inconnue.

2.3 Cthulhu

Votre objectif cette semaine est d'écrire un interpréteur pour le langage *Cthulhu*. C'est une bon entraînement car il est à la fois extrêmement facile à lire et orienté objet. De plus, aucun interpréteur *Cthulhu* n'est connu à ce jour, vous ferez donc partie des rares personnes à pouvoir se vanter d'en avoir écrit un.

Un programme *Cthulhu* est lu ligne par ligne, chaque ligne représentant une fonction *Cthulhu*.

```
1  0A iio
```

Ces lignes sont divisées en trois parties :
- Un nombre naturel compris entre 0 et $+\infty$
- Une lettre majuscule entre A et D.
- Une séquence d'instructions, que nous définirons ci-dessous.

Les deux premières parties sont toujours séparées de la troisième par un seul espace blanc. Une ligne peut également être vide (ce qui signifie qu'il peut y avoir des lignes vides entre deux fonctions).

```
1  0A iii
2
3
4  42B ddd
```

Chaque fonction est donc un objet, fait d'un ID, d'une suite d'instructions et d'un accumulateur initialisé à 0. Dans notre implémentation, une fonction qui n'existe pas n'a pas d'accumulateur (**c'est important !**), et une fonction doit avoir au moins une instruction.

Les instructions que vous pouvez trouver dans la suite d'instructions d'une fonction sont les suivantes :

Caractère	Comportement
i	Incrémente l'accumulateur
d	Décremente l'accumulateur
*	Attend une entrée et la met dans l'accumulateur
o	Affiche l'accumulateur en sortie
[{FunctionID}	Appelle la fonction dont l'ID suit le caractère '['
]{IDLetter}	Appelle la fonction dont la lettre d'identification est IDLetter et dont l'IDNumber est l'accumulateur de cette fonction.
e{FunctionID}	Remplace l'accumulateur de cette fonction par la valeur de l'accumulateur de FunctionID.
E{FunctionID}	Remplace l'accumulateur FunctionID par celui de cette fonction.

Appeler une fonction signifie l'exécuter, ce qui signifie à son tour exécuter toute sa suite d'instructions, de gauche à droite. Enfin, exécuter un programme *Cthulhu* signifie appeler la fonction 0A.

Voici un exemple de programme utilisant toutes les instructions ci-dessus :

```
1  0A *iE1A*E1B[1A
2  1A d]B
3  1B i[1A
4  0B e1Bo
```

0A attends un input(*), le reçoit dans son accumulateur, puis l'augmente de 1(i). Il le place ensuite dans 1A(E1A). Il récupère à nouveau un input(*), le reçoit dans son accumulateur, puis le place de 1B(E1B) et appelle 1A([1A).

1A décrémente son accumulateur de 1(d), puis appelle une fonction de la lettre est B(]B). Pour comprendre comment déterminer quelle est cette fonction, lisez le paragraphe suivant.

1B incrémente son accumulateur de 1(i) puis appelle 1A([1A).

0B reçoit l'accumulateur actuel de 1B(e1B) dans son accumulateur, puis output son accumulateur (o).

Ce programme prends donc deux nombres en input, et output leur addition. Essayez de bien comprendre comment fonctionne ce programme pour comprendre le langage.

Trivial, n'est-ce pas ?

Maintenant, compliquons un peu les choses. Et si nous essayions d'appeler ou d'accéder à l'accumulateur d'une fonction qui n'existe pas ? Et bien dans ce cas, *Cthulhu* suit ce comportement :

1. Choisissez la fonction avec l'IDnumber le plus élevé parmi les fonctions avec la même lettre dont le numéro d'identification est plus petit.
2. S'il n'y en a pas, choisissez la fonction avec la même lettre et l'IDnumber le plus élevé.
3. S'il n'y en a pas, ne faites rien (passez l'instruction).

Cela devrait vous suffire pour apprivoiser ce Grand Ancien. Vous pouvez trouver plus d'informations sur *Cthulhu* sur *esolangs.org*.

2.4 Lambda Expressions et Actions

Si vous avez fait attention en classe, vous vous souvenez peut-être des fonctions **anonymes** d'OCaml :

```
function x -> x + x // anonymous function returning x + x
```

Celles-ci sont très utiles car ce sont des objets qui peuvent être facilement manipulés. Pour ce projet, nous aurons besoin d'objets similaires afin de simplifier la façon dont nous exécutons l'ensemble d'instructions d'une fonction *Cthulhu*.

En effet, pour exécuter une fonction, nous devons parcourir chaque instruction une par une. Par conséquent, en lisant ces instructions, nous voulons pouvoir les stocker dans l'ordre et les exécuter chaque fois que cette fonction est appelée dans notre programme.

Mais comment exprimer et stocker un ensemble d'instructions qui changeront pour chaque fonction ? C'est là que les expressions lambda et Actions entrent en jeu.

2.4.1 En quoi les Actions nous seront-elles utiles ?

Les `Actions` permettent de générer beaucoup de fonctions très simples et au comportement très proches, mais tout de même différentes par les instances qu'elles affectent. De plus, ces objets sont stockables, ce qui signifie qu'au contraire d'une fonction telle que vous les connaissez, il est possible de les mettre dans une liste (plus d'explications dans la partie exercice).

2.4.2 Utilisation et exemples

Les expressions lambda de C# s'écrivent de cette manière :

```
(paramètres) => { déclarations; }
```

Elles peuvent renvoyer une valeur et prendre des arguments, ou non. Dans notre cas, nous nous concentrerons uniquement sur les expressions lambda qui ne renvoient pas de valeur et ne prennent aucun argument.

Afin de stocker une telle expression lambda, nous utilisons des `Actions`. Voici un exemple d'une bonne utilisation des actions avec des expressions lambda :

```
Action hello = () =>
{
    Console.WriteLine("Hello");
}
hello(); // Affiche "Hello" dans la console
```

Comme vous pouvez le voir, elles sont assez simples à utiliser. Voyons maintenant un exemple plus proche de ce que nous aimerions faire.

```
1  public class Counter
2  {
3      private long counter;
4
5      public Counter()
6      {
7          counter = 0;
8      }
9
10     public void Increment()
11     {
12         ++counter;
13     }
14 }
15
16 public class Program
17 {
18     static void main()
19     {
20         Counter c = new Counter();
21         Action twice = () =>
22         {
23             c.Increment();
24             c.Increment();
25         };
26
27         twice(); // Exécute deux fois la méthode Increment() sur c.
28     }
29 }
```

Ici, nous avons conçu une **Action** dans `main` dont le seul but est d'incrémenter c deux fois. Jouez un peu avec pour saisir le plein potentiel de ces objets.

3 You And Cthulhu

3.1 Structure

Dans ce TP, une structure de fichiers a déjà été fournie. Cette section explique ce qui a été créé et écrit dans le squelette donné, pour que vous puissiez continuer ce projet.

Le projet est un interpréteur du langage ésotérique *Cthulhu*. Il est divisé en deux grandes parties :

— Le lexer et parser (l'analyseur), qui prendra le texte donné et le lira ligne par ligne. Le but de cette partie est d'extraire les instructions attendues.

— Une deuxième partie qui gère les instructions données par le parser pour créer un tableau de fonctions et construire le programme.

Vous trouverez également `TestYouAndCthulhu`, dans lequel sont fournis quelques fichiers texte contenant des programmes *Cthulhu* pour tester votre travail, ainsi qu'un interpréteur *Cthulhu* exécutable, présent dans le dossier `Executable`, pour comparer votre résultat pour un programme donné, et le bon résultat. Vous pouvez l'utiliser de la manière suivante :

```
dotnet Executable/YouAndCthulhu.dll PathToYourTestFile
```

3.1.1 LexerParser.cs

La classe `LexerParser` contient déjà une énumération avec tous les tokens de commandes qui peuvent exister. Le constructeur prendra une ligne et la transformera en une `Function` *Cthulhu*. Cet objet sera divisé en deux autres sections : la fonction ID et les commandes. C'est là que nous pouvons détecter si la syntaxe de l'entrée est incorrecte.

3.1.2 Function.cs

La classe `Function` représente une fonction (une ligne) écrite en *Cthulhu*. Cette fonction est composée de :

— Un ID number qui est un nombre naturel.

— Une lettre en majuscule : A, B, C ou D.

— Un registre qui est un nombre signé.

— Une liste d' `Actions` qui correspondent aux instructions qui doivent être exécutées.

Cette dernière partie est ce qui rend les `Actions` intéressantes dans notre cas : nous allons concevoir une `Action` par instruction dans une fonction *Cthulhu*, et l'ajouter à une liste !

3.1.3 FunctionTables.cs

Une instance `FunctionTable` contient l'ensemble des fonctions du programme écrit et permet un moyen plus simple et plus efficace d'accéder aux fonctions. Il est composé d'un tableau de quatre listes de fonctions. Chaque liste correspond à une lettre de A à D (respectivement aux index 0, 1, 2 et 3) et contient les fonctions qui ont pour lettre d'identification la lettre correspondante. Il est également trié par ordre croissant par `id_natural`.

Attention !

Il est strictement **interdit** de modifier le prototype d'une classe ou d'une fonction donnée !

3.2 Fun with Cthulhu

Tip

N'oubliez pas d'écrire des messages dans vos exceptions. Il vous sera plus facile de trouver vos erreurs lors du test de votre code.

3.2.1 LexerParser

Dans cette première fonction, vous recevez une chaîne de caractères comme argument et une référence vers l'index actuel dans la chaîne. À partir de cet index, vous devez extraire un entier positif, mettant à jour l'index. L'index mis à jour est la nouvelle position dans la chaîne, directement après le nombre extrait. Si aucun numéro n'est trouvé, vous devez lancer une exception.

```
1  public static UInt64 GetIdNatural(string s, ref int index)
```

Pour celle-ci, vous devez répéter ce que vous avez fait pour la fonction précédente. Cependant, au lieu d'un nombre, vous devez extraire une lettre et la renvoyer. S'il y a une erreur, vous devez lever une exception.

```
1  public static char GetIdLetter(string s, ref int index)
```

Prenons comme exemple ce string :

1	0	A	' '	i	i
0	1	2	3	4	5

Si on appelle GetIdNatural avec ce string , et l'index 0, la valeur de retour sera **10**, et l'index devient maintenant **2**.
Et maintenant on appelle GetIdLetter avec ce même string, et avec comme index 2, la valeur de retour sera **A** et l'index sera **3**.

Note

Vous pouvez utiliser : `Char.IsDigit()`, `Convert.ToUInt64()`, et toutes les fonctions déjà vues.

La fonction suivante renvoie un `CommandToken`, un token correspondant au caractère à la position `index` dans la chaîne donnée en argument. Si le caractère ne correspond pas à un caractère de commande, la fonction renvoie un `TOKEN_DEFAULT`.

Caractère	CommandToken
i	TOKEN_INCREMENT
d	TOKEN_DECREMENT
*	TOKEN_INPUT
o	TOKEN_OUTPUT
[	TOKEN_CALL
]	TOKEN_CALL_ACC
e	TOKEN_MOVE_IN
E	TOKEN_MOVE_OUT

```
1  public static CommandToken CommandLexer(string s, int index)
```

`LineToFunction()` est déjà donnée. À partir d'une `string` et d'une `FunctionTable`, elle crée une nouvelle `Function` et le renvoie.

```
1  public static void LineToFunction(string s, FunctionTable ftable)
```

3.2.2 Function

Attention !

N'oubliez pas de lire toutes les fonctions avant de commencer à coder cette partie !

Vous commencerez par écrire le constructeur Function. Il doit initialiser ses attributs et remplir `execution` en utilisant `ParseCommands`.

```
1  public Function(UInt64 idnum, char idchar, string commands,
2                  FunctionTable ftable)
```

Note

L'accumulateur est initialisé à 0, dans ce constructeur.
La fonction `ParseCommands` devra être codée à la fin de cette section.

Les méthodes suivantes à implémenter correspondent chacune à une instruction *Cthulhu*, et seront appelées pour "exécuter" une fonction.

Incrémente l'accumulateur.

```
1  public void Increment()
```

Décremente l'accumulateur.

```
1  public void Decrement()
```

Affiche l'accumulateur de la fonction, suivi d'un retour à la ligne.

```
1  public void Output()
```

Input attends un input de l'utilisateur sous forme d'entier signé. Si cet input n'est pas un entier signé, le programme doit renvoyer une `ArgumentException`. Voici le format affiché dans la console en attendant l'input :

```
Waiting for integer input: {PlaceOfTheInput}$
```

N'oubliez pas que le dollar est un retour à la ligne.

```
1  public void Input()
```

Charge l'accumulateur de `f` dans l'instance.

```
1  public void MoveIn(Function f)
```

Charge l'accumulateur de l'instance (cette `Function`) dans `f`.

```
1  public void MoveOut(Function f)
```

Exécute la liste `execution`. Cette méthode consiste tout simplement à exécuter chaque Action. Cela veut dire que vous devez traverser la liste `execution` et pour chaque Action, l'exécuter.

```
1  public void Execute()
```

Cette fonction n'est pas une méthode, mais une fonction de parsing qui a besoin d'être exécutée depuis une instance de `Function`. A partir de la `FunctionTable` et de la `string` contenant les instructions de la fonction, il faut remplir `execution` avec des `Actions`. Cela nécessite à la fois une bonne compréhension des `Actions` et une bonne compréhension de la structure `FunctionTable`, il sera donc plus intéressant de revenir faire cette fonction après avoir fini `FunctionTable`. Selon le token renvoyé par `LexerParser`, on définit une `Action` appropriée et on l'ajoute à la table. Plus d'explication sur les Actions nécessaires se trouvent dans le squelette.

```
1  private void ParseCommands(string commands, FunctionTable ftable)
```

3.2.3 FunctionTable

Le constructeur `FunctionTable` doit initialiser sa table de `Function`.

```
1  public FunctionTable()
```

Puisque ce sujet n'était pas assez amusant, voici maintenant une fonction de recherche dichotomique à implémenter. Celle-ci cherche un ID number dans une `List<Function>`, et renvoie sa position. Si celui-ci n'existe pas, la fonction renvoie la place à laquelle l'ID **DEVRAIT** se trouver. Evidemment, il est interdit d'utiliser une fonction de recherche dichotomique déjà implémentée.

```
1  private int BinarySearch(List<Function> l, ulong idnum)
```

On implémente maintenant la méthode qui ajoute une `Function` à `ftable`. Si une `Function` avec le même ID est déjà présente, la méthode renvoie une exception.

```
1  public void Add(Function f)
```

`Search` et `SearchRegister` retournent toutes deux une `Function`. La première est à utiliser lorsque nous cherchons à partir d'un ID de fonction, tandis que la seconde est utilisé pour une recherche à partir d'un accumulateur (pour l'instruction]). Comme mentionné précédemment, il est possible qu'une fonction cherchée n'existe pas. Dans ce cas, suivez les règles données dans le cours.

```
1  public Function Search(ulong idnum, char idchar)
```

```
1  public Function SearchRegister(int register, char idchar)
```

Cette dernière méthode est triviale : elle exécute (si elle existe) 0A, le "main" de *Cthulhu*. Si elle n'existe pas, la fonction ne fait rien.

```
1  public void Execute()
```

3.2.4 Program

La fin est proche. Vous allez maintenant dompter *Cthulhu*. Cette dernière fonction sera celle à appeler afin de lire un programme Cthulhu contenu dans un fichier. Il s'agit donc d'initialiser une `FunctionTable` et lire le fichier ligne par ligne (comme vous savez désormais si bien le faire), remplir la table, et l'exécuter. N'oubliez pas qu'il peut y avoir des lignes vides !

```
1  public static void CthulhuSlaver(string path)
```

4 Bonus

Si vous atteignez cette partie, votre interpréteur Cthulhu devrait être opérationnel. Que faire maintenant ?

4.1 Interpréteur Deadfish

A partir de votre interpréteur Cthulhu, construisez un interpréteur Deadfish. Comme vous avez pu le voir sur le site d'esolang, un programme et des explications sur la manière dont il devrait fonctionner sont déjà donnés. Attention, ce programme ne marchera pas avec votre interpréteur puisqu'il ne gère pas les registres de fonctions inexistantes. C'est votre rôle de le modifier pour qu'il fonctionne.

Le bonus se présente donc sous la forme d'un fichier "Deadfish" contenant le programme, à la racine du projet.

4.2 Laissez place à votre imagination

Ajoutez plus de fonctionnalités à Cthulhu, tant qu'elles n'empêchent pas à un programme Cthulhu de fonctionner. Présentez chaque addition à votre README.

N'oubliez pas

Au cas où vous implémentez des bonus, expliquez les dans votre README afin d'être sûr que nous ne les oublions pas.

If you knew time as well as I do, you wouldn't talk about wasting it.