Enquête 9 : A Tree-cky Case...

Consignes de rendu

À la fin de ce TP, vous devrez rendre un dépôt Git respectant l'architecture suivante :

```
csharp-tpXX-prenom.nom/
|-- README
|-- .gitignore
|-- Bonus/
    |-- GenTree.ml
    |-- Makefile (si présent)
|-- Warmup/
    |-- Makefile (si présent)
    |-- Warmup.cs
    |-- Program.cs
|-- Whodunit/
    |-- data/
        |-- Everything
    |-- GenTree.cs
    |-- Makefile (si présent)
    |-- Program.cs
    |-- Suspects.cs
```

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

— Remplacez `prenom.nom` par votre propre login.
— Le fichier `README` est obligatoire.
— Pas de fichier de solution `.sln` dans le projet.
— Pas de fichier executable `.exe` dans le projet.
— Pas de dossier `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— Retirez tous les tests de votre code.
— **Le code doit compiler !**

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

Au cours de ce semestre, vous avez utilisé l'IDE JetBrains Rider. Un IDE ("Integrated Development Environment") est un outil puissant conçu pour améliorer votre workflow en mettant à votre disposition des outils essentiels et une interface graphique, mais vous ne l'utiliserez pas cette semaine.

Bien qu'utiliser un IDE soit recommandé dans la plupart des cas, vous n'en aurez pas toujours sous la main et vous pouvez être amené à travailler dans un environnement qui n'en dispose même pas (au hasard, l'an prochain et pendant le reste de vos années à EPITA). Le but de ce TP est donc de vous faire sortir de votre zone de confort et de vous faire découvrir à quel point un simple éditeur de texte peut être puissant et efficace avec un peu de configuration et de pratique.
Vous pouvez voir ce TP comme un aperçu de la façon dont vous allez devoir travailler dans les années à venir !
Nous vous conseillons donc fortement de ne pas utiliser un IDE, ou Thunar pour ce TP et à rester le plus possible dans le terminal.

Au cours de ce TP, vous en apprendrez un peu sur la compliation et la bonne vieille façon de travailler en utilisant Vim et Emacs.

Attention !

Nous vous recommandons très fortement de travailler sur les machines du PIE pour ce TP puisqu'il est attendu que votre code compile sur cet environnement.
Un code qui ne compile pas sera fortement pénalisé. Il est donc crucial pour vous de compiler et de tester votre code.

Une dernière chose...

Afin de nous assurer que vous n'utilisez pas Rider, nous utiliserons une ancienne version de C# pour la correction. Si vous utilisez Rider, vous risquez d'utiliser certaines fonctionnalités qui ne seront pas disponibles dans notre environnement, auquel cas votre TP ne compilerait pas.

2 Cours

Pour comprendre les notions clés de ce TP, il est absolument indispensable de lire l'intégralité du cours avant de commencer.

2.1 Outils pour ce TP

Il s'agit ici de vous présenter les outils que nous voulons que vous utiliserez pour ce TP. Vous connaissez déjà normalement la plupart d'entre eux, mais quelques rappels n'ont jamais fait de mal à personne.

2.1.1 Le shell

Un *shell* est une interface utilisateur, généralement une interface de ligne de commande (ou CLI, pour Command Line Interface) qui permet à un utilisateur d'interagir avec les services du système d'exploitation. En d'autres termes, un shell est la grosse boîte noire dans laquelle vous tapez des commandes et qui les exécute comme par magie. Voici un exemple d'utilisation d'un shell.

```
42sh ~ echo

42sh ~ echo bpl future moulette
bpl future moulette
42sh ~ ls
TPdir evalexpr.c ConfDeJulien.pptx facts.out
42sh ~ cd TPdir
42sh ~ ls
DeuxiemeMeilleurTP TP1 TP2 TP3 TP4 TP5 TP6 TP7 TP8 MeilleurTP
42sh ~ ls ..
TPdir evalexpr.c ConfDeJulien.pptx facts.out
42sh ~ ls -a ..
.evalexprtraces TPdir evalexpr.c ConfDeJulien.pptx facts.out
42sh ~ cd ..
42sh ~ ./facts.out
Plus de 90% des malwares utilisent le registre rax
```

Le texte `42sh~` est ce que nous appelons un **prompt**. Il s'affiche chaque fois que l'utilisateur peut taper une commande dans le terminal.

Vous remarquerez que chaque commande (le texte que l'utilisateur a tapé après le prompt) est construite de la même manière : `programme argument1 argument2 ...`

`programme` est en fait le chemin d'accès au programme que vous voulez exécuter. Il peut s'agir d'un programme intégré (c'est-à-dire qu'aucun fichier exécutable ne lui est attaché et fait partie du programme de l'interpréteur de commandes comme `cd`) ; de fichiers exécutables trouvés dans un chemin par défaut comme `ls` ou `tree` ; ou des fichiers exécutables dans le chemin spécifié, comme le programme `facts` dans notre exemple. Dans ce cas, vous devez taper `./facts.out` pour l'exécuter lorsque vous vous trouvez dans le même répertoire.

Quelques rappels supplémentaires :

— Le fichier le plus élevé de votre *filesystem* est appelé la racine. Sous Unix, son chemin est `/`
— Chaque fichier possède un chemin d'accès à partir de la racine du système de fichiers, appelé le chemin absolu.
— Chaque fichier a également un chemin relatif à la position de tous les autres fichiers. Il s'agit du chemin d'accès relatif. Il s'agit du type de chemin qui ne commence pas par '/'.
— Le répertoire parent d'un fichier se trouve sur le chemin relatif `..` (comme dans notre exemple précédent). D'autre part, `.` se réfère au répertoire courant.
— Les fichiers dont le nom commence par un point sont des fichiers cachés. Vous devez taper des commandes spécifiques pour les voir.

Reminder

En raison de la manière dont le C# est compilé avec csc, gardez à l'esprit que ./Program.exe entraînera une erreur sur le PIE, la manière correcte est d'exécuter avec `mono Program.exe`.

2.1.2 Commandes de bases

Ce sont les commandes les plus basiques du shell que vous devez connaître et être capable d'utiliser :

— `ls` - liste les fichiers
— `pwd` - affiche le dossier actuel
— `cd` - changer de dossier

— `mv` - déplacer un fichier/dossier
— `cp` - copier un fichier/dossier
— `unzip` - dézipper
— `cat` - afficher le contenu d'un fichier
— `touch` - créer un fichier
— `mkdir` - créer un dossier
— `rm` - supprimer un fichier
— `tar` - compresser ou décompresser des fichiers

Tip

Pour toute commande, si vous avez un doute sur son fonctionnement, vous pouvez utiliser `man`, `apropos` ou `whatis` suivi de la commande pour obtenir plus d'informations.

2.2 Editeurs de texte que vous devez connaître

Puisque vous n'utiliserez pas Rider pour ce TP, voici quelques éditeurs de texte que vous pouvez utiliser.

Le premier, que vous connaissez déjà assez bien, est **Emacs**. C'est un outil puissant qui est extrêmement personnalisable. Pour ceux d'entre vous qui ont été traumatisés par OCaml, sachez que **Emacs** est juste un éditeur de texte/code, pas un éditeur OCaml. Vous devriez être en mesure d'utiliser **Emacs** de manière décente maintenant.

La deuxième option, qui est plus répandue à EPITA, est **Vim**. **Vim** est un programme qui s'exécute directement dans votre terminal. Lorsqu'il est bien utilisé, il peut vous permettre de travailler beaucoup plus efficacement, même si c'est un peu difficile au début. L'intérêt principal de **Vim** est qu'il s'agit d'un éditeur de texte *modal* : il possède de nombreux modes d'édition du texte (insérer, remplacer, sélectionner, etc.) qui ont tous leur utilité. Pour apprendre les bases de **Vim**, nous vous recommandons d'utiliser `vimtutor` [1], car il vous expliquera tout ce que vous devez savoir. Pour utiliser vim, tapez simplement `vim` dans le terminal suivi du nom du fichier que vous souhaitez modifier.

Tip

Si vous avez des difficultés à utiliser Emacs ou plus particulièrement Vim, qui n'est pas facile à prendre en main, n'hésitez pas à demander de l'aide à vos ACDC.

2.2.1 Vim

Vous trouverez sur Moodle un fichier nommé **vimrc**. C'est un fichier de configuration utilisé par **Vim** que vous êtes libres de modifier à votre guise. Nous vous proposons une configuration basique mais utilisable si vous ne voulez pas vous prendre la tête avec ça.
Pour l'installer, téléchargez le fichier **vimrc** sur la page Moodle du TP et déplacez le dans le dossier **/afs/.confs/**. Votre **Vim** devrait ensuite utiliser la configuration que nous vous avons donné.

2.2.2 Emacs

Comme pour **Vim**, nous vous proposons une configuration basique pour Emacs. De même, vous êtes libres de la modifier à loisir et il vous suffit de télécharger les fichiers sur la page Moodle du TP et de les mettre dans `/afs/.confs/emacs` pour commencer à les utiliser

1. Il suffit de taper ceci dans un terminal et d'appuyer sur Entrée pour l'exécuter.

2.3 .NET, Mono et la compilation

Vous n'utiliserez pas Rider pour ce TP, cette section est là pour vous apprendre ce que vous devez savoir pour compiler et exécuter votre code.

2.3.1 Compilation vs Interprétation

Une fois que vous avez écrit votre code, vous aurez probablement envie de l'exécuter. Le problème est que votre ordinateur ne parle pas nativement le C#, Python, OCaml, ou tout autre langage que vous utilisez. Votre ordinateur ne parle qu'un seul langage, le *langage machine*, qui est une séquence d'instructions binaires qui n'a pas de sens pour un être humain. Alors comment pourrait-on communiquer avec un ordinateur pour lui dire quelles sont ses instructions ? La réponse est très simple : il faut un traducteur, quelque chose qui traduit votre code lisible par l'homme, *le code source* en *bytecode*, qui peut être executé par l'ordinateur

Il y a deux façons principales de le faire : **compilation** et **interprétation**. La compilation est le processus qui consiste à prendre tout votre code et à le traduire en une seule fois dans le langage machine cible, que vous pourrez ensuite exécuter. D'un autre côté, l'interprétation est un processus qui va exécuter chaque instruction sur votre machine, sans générer de fichier executable.

Chaque stratégie a ses avantages et ses inconvénients.

— La compilation prend du temps, mais une fois qu'ils sont compilés, les programmes ne prennent pas beaucoup de temps pour s'exécuter. D'autre part, l'interprétation ne nécessite pas de temps pour générer un binaire, mais l'interprétation n'est pas aussi rapide.

— La compilation produit un binaire qui ne convient pas à toutes les machines, tandis que le code interprété peut être exécuté sur toutes les machines, puisqu'il ne ne génère pas de binaire. Vous pouvez cependant compiler votre code sur un autre ordinateur (sur une plateforme différente) tout en le gardant exécutable sur votre machine. C'est ce qu'on appelle *cross-compiling*.

Ce ne sont là que quelques exemples des spécificités de la compilation et de l'interprétation. Préférer l'une à l'autre n'est en fait qu'une question de contexte ; il n'y a pas de méthode intrinsèquement meilleure.

2.3.2 Qu'est-ce que .NET

.NET est ce que l'on appelle un framework. Un framework est une sorte de bibliothèque, c'est-à-dire un ensemble de fonctions pré-implémentées qu'un programmeur peut utiliser comme il l'entend. La différence entre une bibliothèque standard et un framework est que ce dernier gère plus que des fonctions prédéfinies, puisqu'il peut il peut traiter du flux de programme, des compilateurs ou d'autres éléments similaires. En outre, un framework peut être étendu ou écrasé par un utilisateur (dans ce cas, le programmeur), mais il ne peut pas être modifié.

.NET est un framework qui a été développé par Microsoft, et est souvent considéré comme étant solide et stable. Il présente un inconvénient de taille : il a été développé principalement pour Microsoft Windows. .NET met en place une machine virtuelle qui gère la sécurité, la gestion de la mémoire et les exceptions.

Lorsque vous construisez votre projet dans Rider à l'aide du framework .NET, c'est compiler votre code dans une bibliothèque[2] qui sera ensuite interprétée à l'aide du framework .NET. L'intérêt de cette opération est qu'elle permet au framework d'interpréter le code écrit dans tous les langages du framework .NET, tels que F# (la version Microsoft d'OCaml) ou Visual Basic.

Par conséquent, lorsque vous développez avec Rider, vous compilez et interprétez en quelque sorte votre code. Pour ce TP, nous souhaitons que vous compiliez uniquement votre code, et à cette fin, nous utiliserons **Microsoft's C# Compiler** (ou `csc` si vous préférez).

2. Une bibliothèque .dll pour être précis.

2.3.3 Qu'est-ce que Mono ?

Comme vous l'avez vu précédemment, le code peut être soit interprété soit compilé, avec des interpréteurs ou compilateurs respectivement. En termes simplifiés, Mono est un projet qui comprend à la fois un compilateur C# et un interprète (ou langage d'exécution commun). Ces deux éléments sont compatibles avec .NET, c'est pourquoi vous verrez souvent ces deux éléments ensemble. Pour cette pratique cependant, vous n'utiliserez que l'interpréteur et vous compilerez avec le compilateur C# de Microsoft.

Comme nous l'avons déjà vu, la compilation du code est distincte de son exécution. En effet, lorsque nous compilons du code, cela crée un fichier binaire que vous pouvez ensuite exécuter. Afin de compiler notre code, nous utiliserons `csc` [3], qui est le compilateur C#. Son utilisation est très simple : il suffit d'utiliser la commande `csc` avec tous les fichiers que vous voulez compiler.

Si vous voulez spécifier le nom du binaire, vous pouvez utiliser la commande `-out:program`, où vous devez remplacer `program` par le nom de votre sortie.

En résumé, si vous avez deux fichiers `Program.cs` et `Basics.cs` que vous voulez compiler en `Basics.exe` vous devez taper :

```
1  csc Program.cs Basics.cs -out:Basics.exe
```

Vous pouvez ensuite exécuter ce binaire en utilisant `mono Basics.exe`

Important

Pour être compilé, votre code doit contenir une méthode `Main`. Si vous n'en avez aucune ou plus d'une, votre code ne compilera pas.

Pour résumer : afin de lancer des programmes C#, il faut compiler son code en un langage intermédiaire, en bytecode, qui sera interpreté par ce qu'on appelle un JIT-compiler [4], qui traduit le bytecode en langage machine a l'exécution.

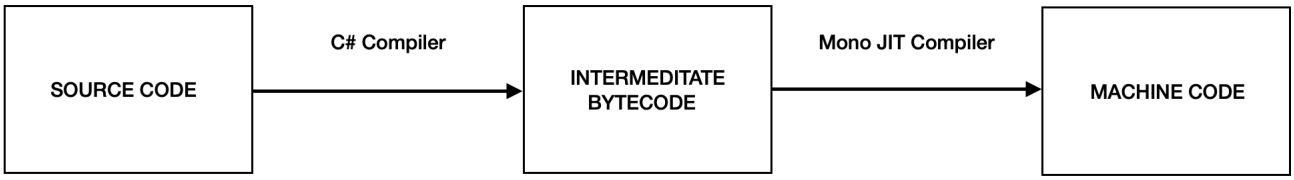

Figure 1 – C#'s compilation pipeline

3. "C-Sharp Compiler".
4. Just-In-Time compiler

2.4 Les fichiers C#

Nous ne vous fournirons pas de squelette pour certains exercices de ce TP et puisque vous n'êtes pas censés utiliser Rider, c'est le moment idéal pour vous expliquer en détail la structure d'un fichier *.cs*.

```
1  using System;
2
3  namespace Basics
4  {
5      public class Hello
6      {
7          public static void Main(string[] argv)
8          {
9              ConfDeJulien();
10         }
11
12         public static void ConfDeJulien()
13         {
14             Console.WriteLine(@"Et la tu reecris ton %eip pour faire une
15             execution de code arbitraire!");
16         }
17     }
18 }
```

L'exemple de code ci-dessus est un fichier *.cs* typique. Nous pouvons voir qu'il est composé de trois types de blocs principaux.

- Le bloc **using**, qui contient des instructions spécifiant quels espaces de noms seront utilisés. Toutes les fonctions de ces namespaces peuvent être utilisées dans le namespace du fichier. Ce bloc est facultatif.
- Le bloc **namespace** qui spécifie le namespace du fichier du fichier. Un namespace est un moyen d'organiser diverses classes en groupes. Une classe appartenant à un namespace différent est donc appelée en spécifiant le premier.
- Le bloc **class** qui contient la définition d'une classe. Vous devriez tous les connaître à ce stade. Il peut y avoir plusieurs définitions de classes dans un même fichier.

Un fichier *.cs* doit respecter cette structure pour être valide et compiler. Un point intéressant, que vous avez peut-être déjà remarqué, est qu'en C# toutes les fonctions font partie d'une classe. En fait, en C#, tout est une classe. Les fonctions sont sont juste des méthodes statiques, ce qui signifie que vous n'avez pas besoin d'instancier un objet pour les appeler.

Important

Veuillez noter que la convention C# consiste à n'avoir qu'une **classe** publique par fichier et de nommer la classe comme le fichier.
Basics.cs contiendra la classe Basics, Complex.cs contiendra la classe Complex et ainsi de suite.

2.4.1 Runtime exception et messages de compilateur

Pendant le dernier TP, vous avez appris ce qu'était les exceptions et comment les utiliser. Néanmoins vous n'avez pas appris à déchiffrer et à comprendre le retour d'un programme ayant levé une exception. D'habitude Rider vous permet d'éviter ce genre de situation, mais pas aujourd'hui, c'est

donc l'occasion parfaite pour apprendre !

D'abord, regardons à quoi ça ressemble :

```
Unhandled Exception:
System.FormatException: Input string was not in a correct format.
  at System.Number.ThrowOverflowOrFormatException (System.Boolean overflow, System.String overflo
  at System.Number.ParseInt32 (System.ReadOnlySpan`1[T] value, System.Globalization.NumberStyles
  at System.Int32.Parse (System.String s) [0x00019] in <12b418a7818c4ca0893feeaaf67f1e7f>:0
  at Example.Program.GetInteger () [0x00006] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.isEven (System.Int32 i) [0x00000] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.Print (System.Int32 input) [0x00000] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.printEvenInput () [0x00007] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.Main () [0x00000] in <3c8fb7b7742940639fc3fafad1418c48>:0
[ERROR] FATAL UNHANDLED EXCEPTION: System.FormatException: Input string was not in a correct form
  at System.Number.ThrowOverflowOrFormatException (System.Boolean overflow, System.String overflo
  at System.Number.ParseInt32 (System.ReadOnlySpan`1[T] value, System.Globalization.NumberStyles
  at System.Int32.Parse (System.String s) [0x00019] in <12b418a7818c4ca0893feeaaf67f1e7f>:0
  at Example.Program.GetInteger () [0x00006] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.isEven (System.Int32 i) [0x00000] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.Print (System.Int32 input) [0x00000] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.printEvenInput () [0x00007] in <3c8fb7b7742940639fc3fafad1418c48>:0
  at Example.Program.Main () [0x00000] in <3c8fb7b7742940639fc3fafad1418c48>:0
```

Figure 2 – Example of stacktrace

Assez indigeste, n'est-ce pas ? N'ayez crainte, c'est très facile à comprendre.

La première chose à remarquer est la deuxième ligne : `Input string was not in a correct format.` C'est le message d'erreurs, celui qui nous dit pourquoi l'execption **System.FormatException** à été renvoyée en premier lieu.
Les quelques lignes suivantes ne nous concernent pas vraiment, il s'agit d'appels à des sous-fonctions internes de C#, nous y reviendront brièvement plus tard.
Ce qui nous intéresse le plus est la ligne `at Example.Program.GetInteger ()` :
C'est la première ligne qui nous parle d'une fonction d'une classe que nous avons implémentée nous-même. Grâce à cette ligne, nous savons que la fonction ayant causé l'erreur et l'arrêt du programme est la **GetInteger** qui ne prend pas d'argument (nous le savons à cause du **()** après le nom de la fonction) appartenant à la classe **Program** du namespace **Example**.
À la lumière de ces informations, nous pouvons aller en chercher un peu plus en regardant la ligne juste au dessus, dans laquelle la présence de `System.Int32.Parse` indique que c'est dans l'appel à cette fonction depuis **GetInteger** que le crash à eu lieu.
Les lignes suivantes sont la suite d'appels de fonctions ayant mené à cette erreur. Dans l'exemple, la fonction `Main` a appelé `printEvenInput`, qui a elle-même appelé `Print`, qui a elle-même appelé `isEven`, d'où l'appel problématique à `GetInteger` à eu lieu.

Voyons à présent les erreurs de compilateurs. Quand vous allez compiler votre code, il y a de grandes chances pour que vous vous trouviez face à des erreurs de compilation (si ce n'est pas le cas, bien joué à vous, vous êtes sur la bonne voie pour devenir des *perfect programmer*). Ce genre d'erreurs apparaît quand le compilateur tente de suivre la *pipeline* décrite plus haut, mais échoue à cause d'une erreur dans votre code.
Imaginez que nous ayons ce fichier, qui affiche sur la sortie standard la devise du meilleur BDE d'Epita :

```
1  public class Program
2  {
3      static void Main()
4      {
5          Console.WriteLine("Votai Test.")
6      }
7  }
```

En essayant de compiler ce programme, **csc** nous renvoie :

```
Program.cs(5,9): error CS0103: The name `Console' does not exist in the current context
Compilation failed: 1 error(s), 0 warnings
```

Figure 3 – Votre première erreurs de compilation !

Mais qu'est-ce à dire que ceci ?
Encore une fois, tout à été pensé pour que ce soit facile à comprendre : Dans le fichier **Program.cs**, à la 9e colonne de la 5e ligne, une erreur a été detectée. Cette erreur est l'utilisation d'une méthode de la classe `Console` qui n'a pas été définie. Cette classe est définie dans **System**, un namespace présent dans le core de C#. Ajoutons le donc à notre programme, nous avons mainteant :

```
1  using System;
2
3  public class Program
4  {
5      static void Main()
6      {
7          Console.WriteLine("Votai Test.")
8      }
9  }
```

Et la sortie du compilateur nous donne :

```
Program.cs(8,5): error CS1002: ; expected
Compilation failed: 1 error(s), 0 warnings
```

Figure 4 – Et la deuxième ! :)

Cette fois, le message est un peu plus étrange : Le compilateur nous indique une erreur à la ligne 8, mais cette ligne est une simple '}'. La vraie erreur est d'avoir oublié un ';' à la fin de la ligne précédente. Après l'avoir corrigée, nous pouvons enfin compiler et exécuter notre programme.

3 Warmup

Cette section n'est là que pour vous permettre de vous familiariser avec l'éditeur et le shell avant d'aborder la partie principale du TP. Elle est également là pour tester votre compréhension du cours que vous avez évidemment lu avant de commencer. Comme vous l'avez peut-être remarqué, il n'y a pas de dossier Warmup dans votre squelette et non, ce n'est pas une erreur. Vous devrez créer le dossier `Warmup` avec un fichier `Warmup.cs` à partir de zéro.
Dans ce fichier, vous n'avez le droit d'utiliser que l'espace de noms System [5] et vous devez définir un espace de noms personnalisé appelé `Warmup` et définir ces exercices dans la classe `Warmup`.

3.1 IsPalindrome

Un palindrome est un mot qui peut être lu à la fois de gauche à droite et de droite à gauche. Vous devez définir une méthode booléenne qui vérifiera si une chaîne de caractères donnée est en fait un palindrome. La chaîne peut comporter n'importe quel type de caractère. Pour cet exercice, nous considérerons comme un palindrome valide uniquement les caractères alphabétiques et nous ne tiendrons pas compte de la case.

```
1 public static bool IsPalindrome(string str) ;
```

```
1 Warmup.IsPalindrome("Anna");                // Renvoie true
2 Warmup.IsPalindrome("rAdAr");               // Renvoie true
3 Warmup.IsPalindrome("rAdAar");              // Renvoie false
4 Warmup.IsPalindrome("k,a;y;;;;;A;;k2930."); // Renvoie true
```

Attention

L'entrée peut être soit vide (ce qui sera considéré comme un argument invalide), soit nulle. Ces deux cas doivent être traités et doivent lever une exception appropriée. Il en va de même des autres exercises.

3.2 RotChar

Vous devez maintenant être familier avec cet exercice. La méthode doit retourner la lettre **c** décalée **key** fois. La clé peut être négative. Vous n'avez à manipuler que des lettres majuscules.

```
1 public static char RotChar(char c, int key) ;
```

```
1 Warmup.RotChar('A', 2);     // Renvoie 'C'.
2 Warmup.RotChar('X', 5);     // Renvoie 'C'.
3 Warmup.RotChar('C', -30);   // Renvoie 'Y'.
```

3.3 RotString

Encode la chaîne **str** en décalant chaque lettre **key** fois.

```
1 Warmup.RotString("IRENE ADLER", -18);   // Renvoie QZMVM ILTMZ
2 Warmup.RotString("CISCO > MIDLAB", 42); // Renvoie SYISE > CYTBQR
```

5. Pour en savoir plus sur les espaces de noms :
https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/namespaces

3.4 Recherche dichotomique

```
public static int BinarySearch(int[] array, int elt);
```

Vous devez implémenter la recherche dichotomique, qui recherche la position d'un élément dans un tableau trié avec une complexité moyenne de $O(\log n)$. Dans le cas échéant, vous devez retourner la position où l'élément devrait se situer. Si le tableau est null, il faut lever une exception appropriée.

```
int[] array = {0,1,3,5,7,8,12};
Warmup.BinarySearch(array, 0);      // Renvoie 0
Warmup.BinarySearch(array, 1);      // Renvoie 1
Warmup.BinarySearch(array, 5);      // Renvoie 3
Warmup.BinarySearch(array, 8);      // Renvoie 5
Warmup.BinarySearch(array, 42);     // Renvoie 7
```

3.5 Makefile (Bonus mais recommandé)

3.5.1 Vue d'ensemble

Comme vous l'avez peut-être remarqué, taper les mêmes commandes chaque fois que vous voulez tester quelque chose ou compiler votre projet devient rapidement très fastidieux. Heureusement, il existe une commande qui peut rationaliser ce processus : **make**.

make est un programme qui permet à l'utilisateur de construire efficacement des projets en définissant un ensemble de règles à exécuter dans ce qu'on appelle un `Makefile`. L'utilisateur peut configurer différentes cibles qui peuvent exécuter différents ensembles de règles en fonction de ses besoins. Les règles sont composées des commandes que l'utilisateur exécute normalement.

```
42sh ~ ls
Makefile Program.cs
42sh ~ make Program
csc Program.cs
Microsoft (R) Visual C# Compiler version 3.6.0-4.20224.5 (ec77c100)
Copyright (C) Microsoft Corporation. All rights reserved.

42sh ~ ls
Makefile Program.cs Program.exe
42sh ~ make run
csc Program.cs
Microsoft (R) Visual C# Compiler version 3.6.0-4.20224.5 (ec77c100)
Copyright (C) Microsoft Corporation. All rights reserved.

mono Program.exe
Hello World !
```

Afin de comprendre la syntaxe d'un Makefile, regardons un exemple simple :

```
variable = OCaml over all

echo :
        echo variable

echovariable :
        echo $(variable)
```

Ici, vous pouvez voir qu'il y a une variable contenant une valeur et deux cibles appelées `echo` et `echovariable`. Les cibles ont toutes deux une règle à l'intérieur, donc quand on appelle make, si aucune cible n'est spécifiée, la première du fichier sera exécutée. La première n'imprimera que le mot variable car pour développer la variable, il faut mettre un signe dollar devant et entre parenthèses (ou accolades).

```
42sh ~ make
echo variable
variable
42sh ~ make echo
echo variable
variable
42sh ~ make echovariable
echo $(variable)
OCaml over all
```

Les règles étant essentiellement des commandes shell, vous êtes libre de mettre ce que vous voulez pour compiler votre projet.

3.5.2 Objectifs

Ecrire un Makefile[6] contenant les règles `compile`, `run` et `clean` qui vont respectivement :
— Compiler le projet
— Compiler **et** exécuter le projet
— Supprimer tous les fichiers qui ont été créés par les règles ci-dessus

Attention

Lorsque vous supprimez des fichiers avec la commande **rm**, il n'y a pas de fenetre de confirmation. Si vous les supprimez **ils sont partis pour de bon**. Il est donc crucial que vous fassiez des **commits** régulièrement votre travail pour pouvoir le récupérer au cas où cela vous arriverait. (Ce n'est vraiment pas drôle quand cela arrive.)

6. Vous voudrez peut-être jeter un coup d'oeil à ce lien :
`https://slashvar.github.io/2017/02/13/using-gnu-make.html`

4 Whodunit ?

Il y a un meurtrier dans la famille...

Sherlock enquête sur une affaire plutôt délicate : une personne a été assassinée dans le manoir familial et personne ne sait ce qui s'est passé. Il est cependant sûr d'une chose : quelqu'un de la famille ment. Sherlock a accès à l'arbre généalogique de la victime et il a besoin de votre aide pour pouvoir faire le tri et rationaliser son travail en mettant en place un moyen facile d'évaluer et de rechercher des suspects dans l'arbre.

Heureusement, notre sociopathe de haut niveau a conçu une implémentation dans ce but précis, mais il a besoin de votre aide pour l'étoffer.

4.1 GenTree

Le détective a structuré son arbre comme suit :

— Chaque noeud représente un membre de la famille et est associé à un nom et à un degré de suspicion.
— Les nœuds de gauche et de droite liés au nœud actuel représentent les parents du membre.
— Tous les membres ont deux parents.

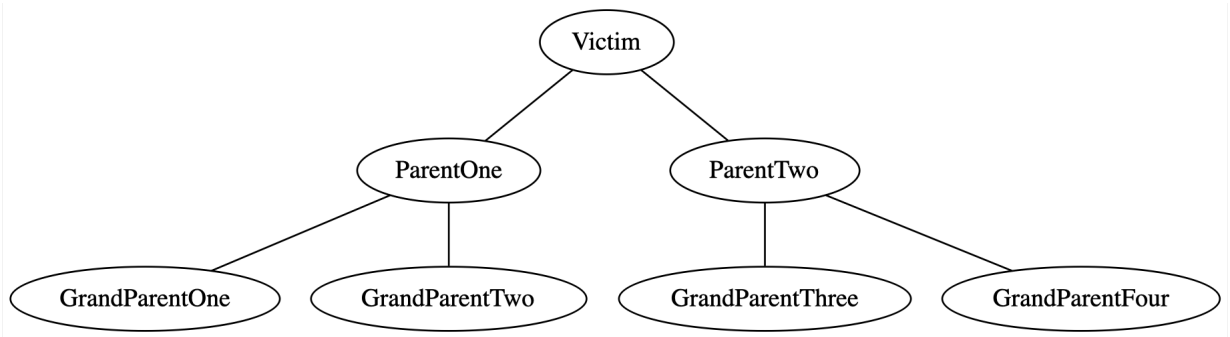

Figure 5 – Un arbre d'ascendance simple

Remarque

Pour simplifier la chose, nous considèrerons que tous les membres de la famille ont des noms distincts.

4.1.1 See what you did there

La première chose à faire est bien évidement un constructeur pour notre classe `GenTree` :

```
1  public GenTree(string name, int suspicion, GenTree left, GenTree right)
```

À présent, Sherlock veut une façon basique de visualiser l'arbre d'ascendance. Il pense que quelque chose comme

```
1  Racine (FilsGauche FilsDroit)
```

serait parfait, et vous demande de l'aider car il est occupé à refléchir et à interroger les suspects.

```
1  public void PrintTree()
```

Par example, en utilisant l'arbre généalogique des Simpson avec Bart Simpson comme racine, il souhaite voir :

```
1  Bart (Homer (Abraham Mona) Marge (Clancy Jacqueline))
```

4.1.2 Change Name

Pour commencer l'affaire, Sherlock a interrogé chaque membre de la famille. Il a ensuite écrit leurs noms et les a placés dans l'arbre. Il y a cependant un problème, il n'est pas sûr de l'orthographe. Il prévoit de demander plus tard à ce que l'orthographe soit modifiée. En attendant, il veut que vous trouviez un moyen de remplacer l'ancien nom par le nouveau dans l'arbre.

```
1  public static bool ChangeName(GenTree root, string oldname, string newname);
```

La méthode doit renvoyer un booléen indiquant si le changement a eu lieu ou non.

4.1.3 Find Path

Le détective consultant a maintenant besoin de savoir exactement comment une personne est liée à une autre dans l'arbre. Il a également besoin que vous gardiez la trace des noms rencontrés entre les deux.

```
1  public static bool FindPath(GenTree root, string name, List<string> path) ;
```

Il ne faut pas ajouter le nom passé en paramètre dans la liste, seulement les noeuds rencontrés lors de la recherche si et seulement si un chemin a été trouvé. Si aucun chemin n'existe, la liste doit être vide. Le chemin se fait du dernier noeud jusqu'à la racine.

4.1.4 Plus petit descendant commun

Le dernier outil nécessaire est celui qui permet à Sherlock de trouver le plus petit descendant commun de deux ancêtres. Le plus petit descendant commun peut être expliqué comme étant le nœud le plus profond de l'arbre dont les deux nœuds sont des ancêtres dans l'arbre.

```
1  public static string LowestCommonDescendant(GenTree root, string PersonA,
2  string PersonB);
```

Par exemple, sur la figure ci-dessus :
— Le plus petit descendant commun entre GrandParentOne et GrandParentTwo est ParentOne
— Le plus petit descendant commun entre GrandParentThree et ParentOne est la Victime.
— Le plus petit descendant commun entre GrandParentFour et ParentTwo est la Victime

4.1.5 To Dot (Bonus)

Lorsque vous travaillez avec des arbres en général, il peut devenir assez compliqué de visualiser la structure elle-même et ce qui se passe lorsque vous la modifiez. Heureusement, il existe un moyen simple d'afficher l'arbre. Cela se fait grâce au langage dot.

Ce langage nous permet d'écrire dans un fichier une représentation de votre graphe (un arbre est simplement un graphe acyclique non dirigé) qui peut être traitée par les programmes [7]. La syntaxe est la suivante :

```
42sh ~ ls
Victim.dot
42sh ~ cat Victim.dot
graph Victim {
Victim -- ParentOne;
Victim -- ParentTwo;
ParentOne -- GrandParentOne;
ParentOne -- GrandParentTwo;
ParentTwo -- GrandParentThree;
ParentTwo -- GrandParentFour;
}
```

Tout d'abord, il y a le type de la structure que vous sauvegardez (qui sera `graph` pour cet exercice). Ensuite, il suffit de mettre le nom du graphe et entre accolades de lister tous les liens entre chaque nœud, en terminant chaque ligne par un point-virgule. Les liens entre deux nœuds dans le graphe non dirigé sont représentés par `--`.

```
public void ToDot();
```

7. N'hésitez pas à regarder ce site pour vérifier votre dot.
`https://dreampuf.github.io/GraphvizOnline`

4.2 Suspects

Dans cette section, Sherlock veut que vous l'aidiez à trier ses suspects en fonction de l'indice de suspicion qu'il a attribué à chacun d'eux. Pour ce faire, il souhaite que vous implémentiez un algorithme de tri décent qui soit très fiable en termes de complexité temporelle dans le pire et le meilleur des cas. Mais d'abord, il est important de transformer l'arbre en tableau.

4.3 De l'arbre au tableau

```
1  public Suspects(GenTree arbre, int taille) ;
```

Cette méthode est en fait le constructeur de la classe `Suspects`. Elle parcourt chaque nœud en utilisant un parcours largeur et insère chaque nœud rencontré dans un tableau de tuples [8] contenant le nom et l'indice de suspicion, le premier étant le seul attribut de la classe.

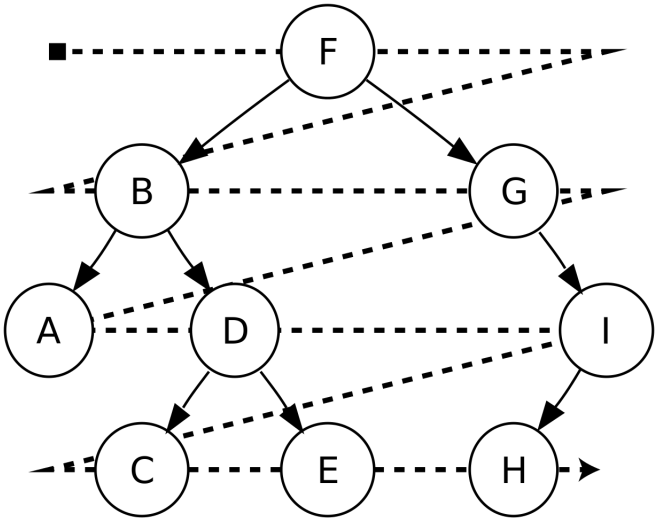

Figure 6 – Traversée de l'arbre par ordre de niveau

8. Plus d'informations ici :
https://docs.microsoft.com/en-us/dotnet/api/system.tuple-2?view=net-5.0

Maintenant que vous avez un tableau de suspects, il est temps de le trier. Pour ce faire, vous allez utiliser l'algorithme de tri par tas.

```
1  public void HeapSort() ;
```

Un Tas est une structure de données qui prend la forme d'un arbre et qui respecte la propriété des tas :

— Si le tas est un tas maximal, la clé du nœud actuel est toujours supérieure ou égale aux clés de ses enfants.

— Si le tas est un tas minimal, la clé du noeud actuel est toujours inférieure ou égale aux clés de ses enfants.

Bien qu'elle soit expliquée comme une structure de données arborescente, elle peut également être représentée comme un tableau unidimensionnel où, pour tout nœud représenté par l'indice i, l'enfant gauche et l'enfant droit sont respectivement situés aux indices $2 * i + 1$ et $2 * i + 2$.

Précision

Pour la suite de l'exercice, nous n'aborderons que les **tas maximaux**.

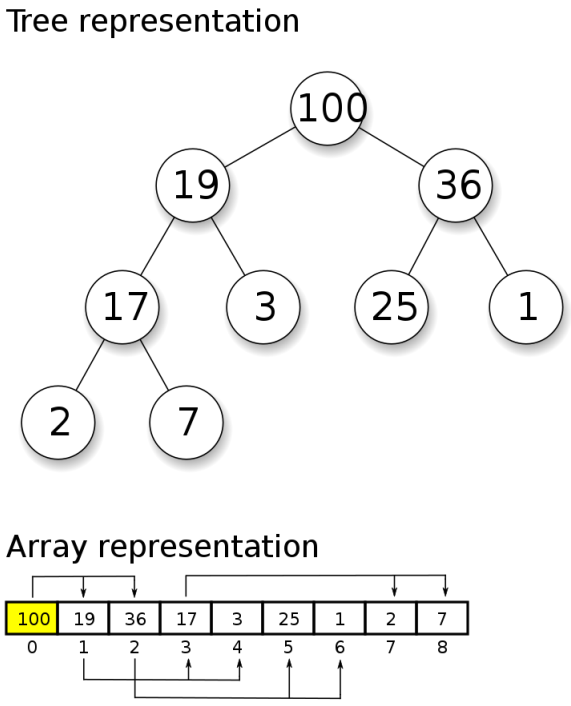

Figure 7 – Représentation d'un même tas en temps qu'arbre binaire et en temps que tableau

L'algorithme Heap Sort est un tri en place qui utilise la propriété de tas, en tamisant le tableau pour respecter la propriété de tas à tout moment. L'objectif de ce tri est de pouvoir toujours récupérer le premier élément (la "racine") et de l'ajouter à la fin du tableau. Les étapes peuvent être décrites comme ceci :

Algorithm 1 : HeapSort

BuildHeap(array) (Transforme un array en tas maximal)
$i \leftarrow length(array) - 1$
while $i >= 0$ **do**
  Échangez la racine du tableau avec l'élément à l'index i
  BuildHeap le sous-tableau de l'index 0 à i
  $i \leftarrow i - 1$
end

4.4 Bonus : Retour vers le S1

Vous l'avez sûrement remarqué, mais les notions de ce TP utilisaient beaucoup la récursion. Vous savez ce qui très adapté à la récursion ? Vous l'avez deviné : OCaml ! À partir du code donné dans le fichier `Bonus/GenTree.ml` vous pouvez ré-implémenter les fonctions :

```
1  val printTree : genTree -> unit = <fun>
2  (* Affiche l'arbre passé comme premier paramètre *)
3  val changeName : genTree -> string -> string -> genTree = <fun>
4  (* Change le nom d'une personne dans l'abre.
5  Le nom à changer est la première string, et il faut la
6  remplacer par la seconde *)
7  val findPath : genTree -> string -> string list = <fun>
8  (* Renvoie une liste de string correspondant au chemin menant à une personne
9  dont le nom est le second  paramètre dans l'arbre passé en premier paramètre.
10 Si aucun chemin n'est trouvé, renvoie une liste vide. *)
```

There is nothing more deceptive
than an obvious fact.