À partir de maintenant, les TP se feront depuis la machine virtuelle "Clef Agrég" installée au début d'année. Si vous ne l'avez pas sur la machine actuelle, télécharger la depuis `http://clefagreg.dnsalias.org` et faites le début sur `https://BetterOCaml.ml/`.

— Pour les exercices sur OCaml, vous utiliserez Emacs et son mode "Tuareg" qui permet d'interpréter le code OCaml ligne par ligne (bloc par bloc, pour être précis) en cliquant dans le menu ou en utilisant les raccourcis clavier. `Ctrl c + Ctrl s` lance un terminal qui s'affiche en bas ou à droite. Attention : au lancement du terminal, il faut bien penser à écrire `ocaml` à la place de `camllight`, puis `Ctrl x + Ctrl e` pour exécuter la phrase (le bloc) actuelle ;

— Pour les exercices en C, je vous demande aussi d'utiliser Emacs et de taper une fois la commande à utiliser pour compiler dans le menu "Tools > compile". Ce sera manuel et peut-être que ça vous semblera laborieux, mais c'est vraisemblablement un environnement proche de ce qui sera disponible aux oraux de concours, donc il faut s'entraîner dans des conditions proches.

Ex1. Réimplémentation maison des listes en OCaml (1h15)

Comme vu en cours, les structures de données de base en OCaml sont les listes et les tableaux. S'il est difficile d'implémenter manuellement des tableaux, il est assez facile de le faire pour des `'a list` et de réécrire manuellement tout un tas de fonctions utiles sur les listes qui sont présentes dans le module `List` de la bibliothèque standard. Sa documentation est en ligne sur `https://ocaml.org/api/List.html`, si besoin.

1.1. Réimplémentation du type de donnée `'a list`

1. Comme vu en cours la semaine dernière, définir un type de donnée `'a list` (un type énumération, paramétrique et récursif) qui permette de définir `[]` la liste vide et `tete :: queue` la concaténation d'une tête `tete` et d'une queue de liste `queue`.
2. Définir quelques exemples de listes et les stocker dans des variables, pour écrire plus rapidement des tests par la suite.

1.2. Réimplémentation des fonctions sur les listes

Pour chacune des fonctions suivantes qui sont dans le module `List`, écrivez votre propre version en suivant la spécification, c'est à dire son typage et sa documentation.

3. `val hd : 'a list -> 'a` : *Return the first element of the given list. Raise Failure hd if the list is empty.* Vous pouvez utiliser un filtrage (`match liste with | head :: _ -> ...` par exemple) ou une déconstruction manuelle (`let head :: _ = liste in ...` par exemple).
4. `val tl : 'a list -> 'a list` : *Return the given list without its first element. Raise Failure tl if the list is empty.* Vous pouvez utiliser un filtrage (`match liste with | _ :: tail -> ...` par exemple) ou une déconstruction manuelle (`let _ :: tail = liste in ...` par exemple).
5. `val nth : 'a list -> int -> 'a` : *Return the n-th element of the given list. The first element (head of the list) is at position 0.*

Les faire toutes prend du temps, vous finirez à la maison comme exercice à emporter.

6. `val rev : 'a list -> 'a list` : *List reversal.* Transforme `[a1; a2; ...; an]` en `[an; an-1; ...; a2; a1]`.
7. `val append : 'a list -> 'a list -> 'a list` : *Concatenate two lists.* Vous pouvez utiliser la fonction précédente `rev`.
8. `val iter : ('a -> unit) -> 'a list -> unit` : *iter f [a1 ; ... ; an] applies function f in turn to a1 ; ... ; an. It is equivalent to begin f a1 ; f a2 ; ... ; f an ; () end.* `iter f liste` consiste à appliquer la fonction `f : 'a -> unit` successivement à toutes les valeurs de la liste. Par exemple `let affiche_liste_entier liste = List.iter (fun x -> print_int x; print_string " -> ") liste` définie une fonction `affiche_liste_entier : 'a list -> unit` qui affiche une liste d'entier comme cela :

```
# let affiche_liste_entier = List.iter (fun x -> print_int x; print_string " -> ") ;;
val affiche_liste_entier : int list -> unit = <fun>
# affiche_liste_entier [1; 2; 3; 4] ;;
1 -> 2 -> 3 -> 4 -> - : unit = ()
```

10. `val map : ('a -> 'b) -> 'a list -> 'b list` : `List.map f [a1; ...; an]` *applies function* `f` *to* `a1, ..., an`, *and builds the list* `[f a1; ...; f an]` *with the results returned by* `f`.
11. `val for_all : ('a -> bool) -> 'a list -> bool` : `List.for_all p [a1; ...; an]` *checks if all elements of the list satisfy the predicate* `p`. *That is, it returns* `(p a1) (p a2) ... (p an)`.
12. `val exists : ('a -> bool) -> 'a list -> bool` : `List.exists p [a1; ...; an]` *checks if at least one element of the list satisfies the predicate* `p`. *That is, it returns* `(p a1) || (p a2) || ... || (p an)`.
13. `val mem : 'a -> 'a list -> bool` : `mem a li` is `true` if and only if `a` is equal to an element of `li`.
14. `val find : ('a -> bool) -> 'a list -> 'a` : `find p li` *returns the first element of the list* `li` *that satisfies the predicate* `p`. *Raise* `Not_found` *if there is no value that satisfies* `p` *in the list* `li`.

1.3. Compilation de code OCaml

Vous pouvez essayer de compiler votre fichier OCaml avec une des lignes de code suivantes (si votre code OCaml est sauvegardé dans un dossier `ce_dossier/`, sous le nom `TP5_ocaml.ml`) :

— compilation vers le *bytecode* OCaml :

```
$ cd ce_dossier/
$ ocamlc -o TP5_ocaml.ocamlrun TP5_ocaml.ml
$ ./TP5_ocaml.ocamlrun
```

— compilation vers du binaire natif :


```
$ cd ce_dossier/
$ ocamlopt -o TP5_ocaml.ocamlopt TP5_ocaml.ml
$ ./TP5_ocaml.ocamlopt
```

Les deux binaires produiront le même résultat (par exemple des affichages d'exemples si vous en avez fait, ou rien si vous avez fait vos exemples dans des `assert ...`), mais le premier s'exécute en interprétant un code binaire intermédiaire (le bytecode OCaml, multi-plateforme) par une machine virtuelle appelée `ocamlrun`, et le second est du binaire machine indépendant de OCaml mais dépendant donc de votre architecture (ici, GNU/Linux via la machine virtuelle ClefAgreg lancée avec VirtualBox).

Ex2. Compilation en C (45 minutes)

Vous commencerez à partir d'un fichier `TP5.c` contenant le texte suivant, que vous recopierez dans Emacs :

```
#include <stdio.h>
#include <math.h>

int main(int argc, char* argv[]) {
    // TODO remplir ici
    printf("%i", 2000+21);
    return 0;
}
```

et la commande suivante pour le compiler (-O0 "-grand o zéro" est le premier argument) puis l'exécuter :

```
$ gcc -O0 -lm -Wall -Werror -o TP5.exe TP5.c
$ ./TP5.exe
2021
$ ...
```

1. Faire des calculs numériques sur les types de bases que sont les entiers (`int`), les flottants simple précision (`float`) ou double précision (`double`) ;
2. En important `#include <stdbool.h>` on a aussi accès à `true` et `false` de type `bool`. Ce sont des alias pour `1` et `0` : en C tout ce qui s'évalue à `0` est faux et le reste est vrai ;
3. L'argument `-lm` donné à `gcc` permet de lier la bibliothèque mathématiques, importée avec `#include <math.h>` en en-tête. Tester quelques fonctions de bases comme `sqrt`, `sin`, `atan` etc. Avec `atan`, définir une constante `pi` $= \pi$ et l'afficher avec un `printf("%f", pi);` ;
4. La fonction `printf` permet des affichages sous la forme `printf("chaine motif", valeur1, ..., valeurN)`. La chaîne motif peut contenir des drapeaux de la forme `%i` (pour des `int`), `%f` ou `%g` (pour des `float`), `%c` (pour des `char`), `%s` (pour des `char[]` qui sont des chaînes de caractères). Ces drapeaux voient leurs valeurs remplacées par les valeurs données dans la suite de la fonction `printf` (c'est comme en Python avec `"chaine motif" % (arg1, ..., argN)` ou les f-string ou le `string.format(arg1, ..., argN)`).

5. Expliquer ce que fait le programme suivant, et expérimenter un peu avec :

```
#include <stdio.h>
#include <math.h>

int main(int argc, char* argv[]) {
    for (int i = 0; i < argc; i = i + 1) {
        printf("Argument numéro %i = %s", i, argv[i]);
    }
    return 0;
}
```

et la ligne de commande suivante pour le compiler et la suivante pour l'exécuter :

```
$ gcc -O0 -lm -Wall -Werror -o TP5.exe TP5.c
$ ./TP5.exe arg1 "arg2 en plusieurs mots" arg3 arg4 "arg5 ?"
```

6. Lors de la compilation, l'argument `-Wall` active tous les avertissements (*warnings*), et l'autre option `-Werror` déclare tous les avertissements comme des erreurs, donc ils vont empêcher la compilation en cas d'advertisement. C'est l'option la plus sûre. Si vous rencontrez des avertissements ou des erreurs lors de la compilation, faites comme avec OCaml : il faut bien lire les messages d'erreurs et essayer de les comprendre pour se débloquer. Vous pouvez aussi appeler l'aide le professeur en charge du TP.

