## Exercice 1. Expressions bien parenthésées

Soit `n` $\in \mathbb{N}$ et `ouvr`, `ferm` deux tableaux de taille `n` contenant des caractères. On suppose que les éléments de `ouvr` (resp. `ferm`) sont distincts deux à deux et que ces deux tableaux n'ont pas d'élément en commun. Par exemple :

```
let ouvr0 = [| '('; '['; '<'; '{' |];;
let ferm0 = [| ')'; ']'; '>'; '}' |];;
```

L'ensemble des expressions bien parenthésées noté $\mathcal{E}$ est le plus petit ensemble tel que :

($P_1$) Si `s` est une chaîne de caractères ne contenant aucun élément de `ouvr` ou de `ferm`, alors `s` $\in \mathcal{E}$.

($P_2$) Si `s` $\in \mathcal{E}$ et `t` $\in \mathcal{E}$, alors `s` ^ `t` $\in \mathcal{E}$ où ^ désigne l'opérateur de concaténation.

($P_3$) Pour tout `s` $\in \mathcal{E}$ et tout `i` $\in [\![0, n-1]\!]$, si on note :
- → $s_1$ la chaîne de caractères dont l'unique caractère est `ouvr.(i)`.
- → $s_2$ la chaîne de caractères dont l'unique caractère est `ferm.(i)`.

alors $s_1$ ^ `s` ^ $s_2 \in \mathcal{E}$.

Par exemple, les expressions suivantes sont bien parenthésées :

⋆ En utilisant ($P_1$) :

```
"let a = "    "1+2"    "*"    "3+4"    "-8"    " in "    "a; a+1; a+2"    ";;"
```

⋆ En utilisant ($P_3$) :

```
"(1+2)"              "(3+4)"              "[a; a+1; a+2]"
```

⋆ En utilisant ($P_2$) et ($P_3$) :

```
"(3+4)-8"     "((3+4)-8)"     "let a = (1+2)*((3+4)-8) in [a; a+1; a+2];;"
```

En revanche les expressions suivantes ne sont pas bien parenthésées :

```
"(1+2)*)4"          "5+(4*3-(2)"          "10*(7+5]"          "]1; 2; 3["
```

Étant donné un caractère `c` du tableau `ouvr` (resp. `ferm`), on veut pouvoir déterminer en temps constant l'élément associé à `c` dans `ferm` (resp. `ouvr`). Pour cela, on utilisera deux tables de hachage « `d_ouvr: corresp` » et « `d_ferm: corresp` » où :

```
type corresp = (char, char) Hashtbl.t;;
```

Par exemple, avec `ouvr` et `ferm` les deux tableaux donnés au début de l'énoncé :

```
# Syntaxe Python:
d_ouvr0 = { '(' : ')' , '[' : ']' , '<' : '>' , '{' : '}' }
d_ferm0 = { ')' : '(' , ']' : '[' , '>' : '<' , '}' : '{' }
```

1. Écrire une fonction

   ```
   make_corresp: char array -> char array -> (corresp * corresp)
   ```

   qui prend en entrée `ouvr` et `ferm`, et renvoie `d_ouvr` et `d_ferm`. On utilisera les fonctions du module `Hashtbl` décrites en cours.

2. À l'aide d'une pile, écrire une fonction

   ```
   est_bien_par: corresp -> corresp -> string -> bool
   ```

   qui prend en entrée `d_ouvr`, `d_ferm`, ainsi qu'une chaîne de caractères `s` et indique si `s` est bien parenthésée. Votre fonction devra être de complexité linéaire en la taille de `s`. Testez avec :

   ```
   "(1+2)*)4"          "5+(4*3-(2)"          "10*(7+5]"          "]1; 2; 3["
   ```

## Exercice 2. Implémentation de dictionnaires

Dans cet exercice, on va implémenter la structure de dictionnaire de trois façons différentes. Les dictionnaires contiendront des objets de type « `'a elem_dict` » composés d'une clé de type `int` et d'une valeur de type `'a` :

```
type 'a elem_dict = {cle: int; valeur: 'a};;
```

On programmera quatre opérations sur les dictionnaires :

→ `create` qui créé un dictionnaire vide.

→ « `add: 'a dict -> 'a elem_dict -> unit` » qui ajoute un élément au dictionnaire.

→ « `find: 'a dict -> int -> 'a` » qui renvoie la valeur de l'élément dont la clé est donnée en argument.

→ « `remove: 'a dict -> int -> unit` » qui supprime l'élément dont la clé est donnée en argument.

**Implémentation 1.** On souhaite implémenter la structure de dictionnaires à l'aide d'une ***table de hachage*** et gérer les collisions par ***chaînage*** (dont le principe a été décrit dans le cours). Les éléments du dictionnaire seront donc stockés dans un tableau de taille $m$. La case d'indice $i$ du tableau contient la liste chaînée de tous les éléments dont la clé $c$ vérifie $h(c) = i$. On utilisera la fonction de hachage :

$$h : c \mapsto \left\lfloor m \times c\frac{\sqrt{5}-1}{2} \right\rfloor \mod m$$

Remarque : lorsqu'on ajoute un élément `e` dont la clé appartient déjà au dictionnaire, on se contentera d'ajouter `e` dans la liste appropriée sans supprimer l'élément de même clé qui appartenait déjà au dictionnaire. Cela signifie que si on ajoute successivement deux éléments `e1` et `e2` de clé $c$ (avec la fonction `add`), puis qu'on supprime l'élément de clé $c$ (avec la fonction `remove`), alors le dictionnaire contient encore l'élément `e1`. Notez que les objets de type `Hashtbl` en OCaml fonctionnent de cette manière.

1. Définir un type « `'a dict1` » et écrire les fonctions `create_d1`, `add_d1`, `find_d1` et `remove_d1`. La fonction `create_d1` prendra en argument la taille du tableau à créer.
2. Évaluer les complexités de ces fonctions.

**Implémentations 2 et 3.** Voir l'exercice à faire pour la semaine prochaine.

## Exercice 3. Files de priorité

Une file de priorité de type « `'a file_prio` » permet de stocker des éléments de type « `'a elem_fp` » composés d'une valeur de type `'a` et d'une priorité de type `int` :

```
type 'a elem_fp = {valeur: 'a; prio: int};;
```

C'est une structure des données dynamique et mutable dans laquelle les éléments sont extraits en commençant par celui ayant la priorité la plus élevée :

(a) Créer une file de priorité vide.

| Priorité | |
|---|---|
| Valeur | |

(b) Insérer `"DS Physique"` avec priorité `12`

| Priorité | 12 |
|---|---|
| Valeur | `"DS Physique"` |

(c) Insérer `"DM Maths"` avec priorité `10`

| Priorité | 12 | 10 |
|---|---|---|
| Valeur | `"DS Physique"` | `"DM Maths"` |

(d) Insérer `"Rendu ITC"` avec priorité `20`

| Priorité | 12 | 10 | 20 |
|---|---|---|---|
| Valeur | `"DS Physique"` | `"DM Maths"` | `"Rendu ITC"` |

(e) Extraction $\rightsquigarrow$ renvoie `"Rendu ITC"` (élément de priorité maximale)

| Priorité | 12 | 10 |
|---|---|---|
| Valeur | `"DS Physique"` | `"DM Maths"` |

(f) Insérer `"Rendu OPT"` avec priorité `23`

| Priorité | 12 | 10 | 23 |
|---|---|---|---|
| Valeur | `"DS Physique"` | `"DM Maths"` | `"Rendu OPT"` |

(g) Modifier la priorité de `"DM Maths"` en `50`

| Priorité | 12 | 50 | 23 |
|---|---|---|---|
| Valeur | `"DS Physique"` | `"DM Maths"` | `"Rendu OPT"` |

(h) Extraction $\rightsquigarrow$ renvoie `"DM Maths"`

| Priorité | 12 | 23 |
|---|---|---|
| Valeur | `"DS Physique"` | `"Rendu OPT"` |

On implémentera quatre opérations sur les files de priorité :

→ `create` qui créé une file de priorité vide.

→ « `is_empty: 'a file_prio -> bool` » qui teste si une file de priorité est vide.

→ « `insert: 'a file_prio -> 'a elem_fp -> unit` » qui ajoute un élément à une file de priorité.

→ « `pull: 'a file_prio -> 'a elem_fp` » qui supprime un élément de priorité maximale de la file et renvoie cet élément. Dans le cas où plusieurs éléments sont possibles, la fonction `pull` choisira arbitrairement l'élément à renvoyer parmi ceux de priorité maximale.

**Implémentation 1.** On souhaite implémenter les files de priorité à l'aide d'une liste qui sera maintenue triée par ordre décroissant des priorités : l'élément avec la plus grande priorité est au début de la liste et celui dont la priorité est minimale est à la fin de la liste.

1. Définir un type « `'a file_prio1` » pour représenter
2. Écrire les fonctions `create_fp1`, `is_empty_fp1`, `insert_fp1` et `pull_fp1`. La fonction pour créer la file vide doit être de type « `unit -> 'a file_prio1` ».
3. Évaluer les complexités de ces fonctions.

**Implémentation 2.** On choisit d'implémenter les files de priorités en utilisant un tableau. Ce tableau sera maintenu trié par ordre croissant des priorités.

4. Définir un type « `'a file_prio2` » pour représenter une file de priorité à l'aide d'un tableau.

Étant donné qu'un tableau a une taille fixe, la fonction `create` doit prendre en argument la taille maximale de la file de priorité (qui est également la taille du tableau).

5. Écrire les fonctions « `create_fp2: int -> 'a file_prio2` » et `is_empty_fp2`. Si besoin, on pourra rajouter des arguments à la fonction `create_fp2`.

Pour ajouter un élément à la file de priorité, on l'insérera dans le tableau en décalant tous les éléments dont la priorité est supérieure. De plus, on calculera la position de l'élément dans le tableau à l'aide d'une recherche dichotomique.

6. Écrire les fonctions `insert_fp2` et `pull_fp2`.
7. Évaluer les complexités de ces fonctions.

## Exercice 4. Implémentation de dictionnaires (suite)

> Relire les explications données au début de l'exercice 2 et revoir le type « `'a option` » décrit en cours.

**Implémentation 2.** Dans cette implémentation, on va stocker les éléments du dictionnaire dans un tableau de taille $m$. Une case du tableau contiendra « `None` » ou bien « `Some e` » où « `e: 'a elem_dict` » :

```
type 'a dict2 = {
    tab: 'a elem_dict option array;
    mutable nb_elem: int;
  };;
```

Le champ `nb_elem` contient le nombre d'éléments dans le dictionnaire, il faut donc penser à le maintenir à jour. Les éléments du dictionnaire seront stockés dans les cases de `tab` indicées de `0` à `nb_elem - 1` ; les autres cases contiendront `None`.

1. Écrire une fonction « `create_d2: int -> 'a dict2` » qui prend en entrée $m$ et renvoie un dictionnaire vide.

Pour ajouter un élément `e` au dictionnaire `d`, on commence par vérifier si la clé $c$ de `e` apparaît dans `d`. Si c'est le cas, on remplace l'ancien élément de clé $c$ par `e`. Dans le cas contraire, on stocke `e` dans la première case contenant `None`.

2. Écrire la fonction « `add_d2: 'a dict2 -> 'a elem_dict -> unit` ».
3. Écrire la fonction « `find_d2: 'a dict2 -> int -> 'a elem_dict` ».
   **Consigne :** il est interdit de faire du copier/coller à partir du code de la question précédente : créez des fonctions intermédiaires et utilisez les dans `add_d2` et `find_d2`. Si vous ne respectez pas cette consigne, votre fonction sera considérée comme fausse.

Afin de supprimer un élément de clé $c$ du dictionnaire, on repère l'indice $i$ de cet élément dans le tableau, puis on décale tous les éléments d'indice $j > i$.

4. Écrire la fonction « `remove_d2: 'a dict2 -> int -> unit` ».
   **Consigne :** le copier/coller est interdit.
5. Évaluer les complexités de ces fonctions.

**Implémentation 3 (facultatif).** On souhaite implémenter la structure de dictionnaires à l'aide d'une ***table de hachage*** et gérer les collisions par ***adressage ouvert***. Les éléments du dictionnaire seront stockés dans un tableau de taille $m$ en utilisant la fonction de hachage :

$$h : c \mapsto \left\lfloor m \times c\frac{\sqrt{5}-1}{2} \right\rfloor \mod m$$

Un élément de clé $c$ est donc stocké dans la case d'indice $h(c)$ qui contient un seul élément (il n'y a pas de liste chaînée dans cette implémentation). La gestion de collisions sera faite par ***adressage ouvert***. C'est à dire que si la case d'indice $h(c)$ est déjà occupée, l'élément est stocké dans la première case inoccupée parmi les cases $(h(c)+1 \mod m)$, $(h(c)+2 \mod m)$ ... Dans le cas où le tableau est plein, un appel à la fonction `add` déclenche une erreur. De plus, si on ajoute un élément dont la clé appartient déjà au dictionnaire, l'élément déjà présent dans le tableau est remplacé. Enfin, on suppose qu'on ne peut pas supprimer d'élément du dictionnaire (en particulier, on n'implémentera pas la fonction `remove`).

6. Définir un type « `'a dict3` » et écrire les fonctions `create_d3`, `add_d3` et `find_d3`. La fonction « `create_d3 : int -> 'a dict3` » prendra en argument la taille du tableau à créer.
7. Évaluer les complexités de ces fonctions.