TP n°13 - I/O OCaml et complexité amortie

Les fonctions d'entrées-sorties OCaml fonctionnent sur le même principe que les fonctions I/O en langage C. La communication avec les périphériques (disque dur, écran, clavier...etc) est faite par l'intermédiaire de canaux.
En OCaml, les deux **types** suivants permettent de représenter des canaux.

`in_channel` **:** type canal entrant, c'est-à-dire un canal qui servira à acheminer des données depuis un périphérique vers le processus de votre code en cours d'exécution

`out_channel` **:** type canal sortant, c'est-à-dire un canal qui servira à acheminer des données vers un périphérique depuis le processus de votre code en cours d'exécution

Pour rappel, en C, il n'y avait qu'un seul type `FILE *` pour représenter un canal.

Comme en C, les trois constantes suivantes correspondent aux canaux standards d'entrée, de sortie et d'erreur selon la norme POSIX :

`stdin` **:** objet canal entrant standard, de type `in_channel`, permettant par défaut d'acheminer des données depuis le clavier vers le processus

`stdout` **:** objet canal sortant standard, de type `out_channel`, permettant par défaut d'acheminer des données vers l'écran depuis le processus

`stderr` **:** objet canal sortant standard pour les erreurs, de type `out_channel`, permettant par défaut d'acheminer des informations d'erreur vers l'écran depuis le processus

Toutes les fonctions de la bibliothèque standard OCaml qui permettent d'acheminer des données depuis et vers les périphériques manipulent des objets de type canaux comme ceux décrits ci-dessus.

Ouverture d'un canal entrant : `open_in : string -> in_channel`
Cette fonction ouvre le fichier s'il existe et déclenche l'exception `Sys_error` sinon.

Ouverture d'un canal sortant : `open_out : string -> out_channel`
Cette fonction crée le fichier indiqué s'il n'existe pas ou l'écrase s'il existe.

Fermeture d'un canal entrant : `close_in : in_channel -> unit`

Fermeture d'un canal sortant : `close_out : out_channel -> unit`

Lecture depuis un canal entrant général. La seule fonction au programme est :

```
input_line : in_channel -> string
```

Elle lit des caractères depuis le canal entrant jusqu'au prochain caractère de retour à la ligne. Elle retourne la chaîne de caractères lue, sans le caractère de retour à la ligne de la fin.

Écriture vers un canal sortant général. La seule fonction au programme est :

```
output_string : out_channel -> string -> unit
```

Elle écrit un chaîne de caractères donnée en deuxième argument d'entrée de la fonction dans le canal sortant donné en premier argument d'entrée.

Lecture depuis l'entrée standard (par défaut le clavier)
La fonction `Scanf.scanf` calquée sur celle du langage C, existe en OCaml, avec la même syntaxe.
Mais on dispose aussi de fonctions propres au langage OCaml, parfois plus simples à utiliser :

`read_int : unit -> int :` lecture d'un entier tapé au clavier, la fonction de conversion `int_of_string` étant utilisée par `read_int` pour convertir la chaîne tapée en entier.

`read_float : unit -> float :` lecture d'un flottant tapé au clavier, la fonction de conversion `float_of_string` étant utilisée par `read_float` pour convertir la chaîne tapée en flottant.

`read_line : unit -> string :` lit les caractères tapés au clavier par l'utilisateur jusqu'à rencontrer un caractère de retour à la ligne. La fonction retourne la chaîne de caractères lue sur l'entrée standard, sans le caractère de retour à la ligne.

Quand une fin de fichier est rencontrée l'exception `End_of_file` est déclenchée.

Écriture vers la sortie standard (par défaut l'écran) Pour écrire vers la sortie standard, généralement l'écran, on dispose là aussi de nombreuses fonctions. Nous avons vu la fonction `Printf.printf` calquée sur celle du langage C. Mais il y a des fonctions propres au langage OCaml pour des affichages simples :

`print_int : int -> unit :` affichage d'un entier à l'écran. Cette fonction utilise la fonction de conversion `string_of_int`.

`print_float : float -> unit :` affichage d'un flottant à l'écran en notation décimale. Cette fonction utilise la fonction de conversion `string_of_float` : l'affichage peut ne pas être aussi précis que la réalité.

`print_char : char -> unit :` affichage d'un unique caractère

`print_string : string -> unit :` affichage d'un chaîne de caractère suivie d'un retour à la ligne

`print_endline : string -> unit :` quasiment identique à `print_string`, mais en forçant ensuite la vidange [1] du canal de sortie standard pour éviter tout délai dans l'affichage.

`print_newline : unit -> unit :` affiche un caractère de retour à la ligne à l'écran et vidange le canal de sortie standard.

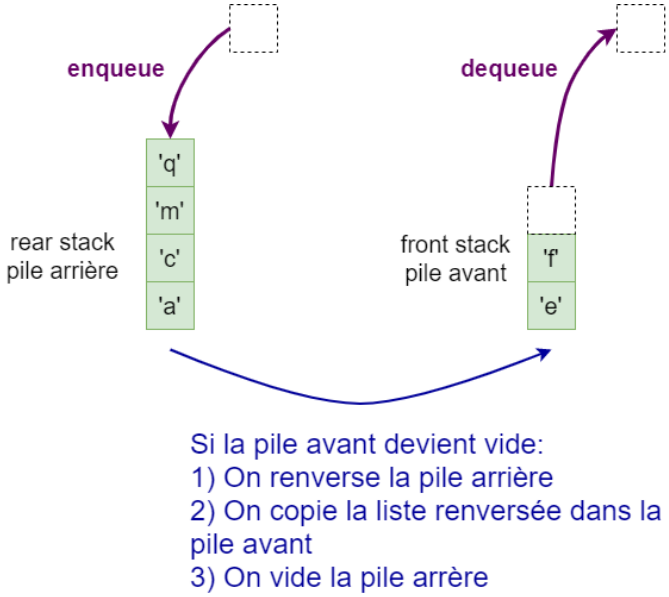
Exercice 1 (Manipulation des fonctions I/O OCaml vers et depuis les canaux standard).
Nous avons déjà codé en C le jeu suivant : un maître du jeu choisit secrètement un entier aléatoire entre 0 et 100 (inclus). Le joueur cherche à deviner ce nombre. A chaque essai, le maître du jeu lui indique s'il a trouvé le nombre mystère et, si ce n'est pas le cas, lui indique si le nombre mystère est plus grand ou plus petit que celui proposé. Le jeu s'arrête lorsque le joueur a trouvé le nombre mystère. Le but du jeu est bien sûr de trouver le nombre mystère avec le moins d'essais possibles. A la fin du jeu, le maître du jeu indique au joueur le nombre d'essais qu'il a dû faire avant de trouver le nombre mystère.

Coder ce jeu de devinette dans un fichier source `NOM_devinette.ml`, la machine jouant le rôle de maître du jeu et l'utilisateur le rôle du joueur. Pour que le jeu soit interactif, l'interprétation du script OCaml sera lancée en ligne de commande sans argument, et non exécutée pas à pas avec le module Tuareg comme nous en avons pris l'habitude. On appliquera les principes de la programmation défensive en essayant de récupérer les exceptions des fonctions d'I/O avec des syntaxes `try...with`, cf cours de la Séquence 5 sur la gestion des exceptions.

Exercice 2 (Lecture depuis un fichier, écriture vers un fichier).
Écrire un code qui lit un fichier texte et le réécrit dans un autre fichier en doublant tous les e.
Vous pourrez vous entraîner sur les fichiers textes utilisés pour le TP 10, qui sont toujours disponibles sur Moodle.

1. En anglais, on parle de *flush*, de chasse-d'eau !

Exercice 3 (File immuable en OCaml à l'aide de deux piles).
Dans un fichier `queue-immutable-two-stacks.ml`, implémenter en OCaml un structure de données de file immuable et polymorphe en utilisant la technique des deux piles avant et arrière vue en cours.



Vous devrez implémenter les 4 opérations classiques inhérentes à la structure de données abstraite de file :

- `queue_create`
- `queue_enqueue`
- `queue_peek`
- `queue_dequeue`

Vous vous convaincrez de l'immuabilité des files manipulées avec cette implémentation, en faisant de nombreux tests.

Exercice 4 (Analyse de complexité amortie).

1. Copiez le code de l'exercice 1 `queue-immutable-two-stacks.ml` dans un nouveau fichier nommé `complexite-amortie.ml`. **Il est interdit d'utiliser l'interface pour faire ce genre d'opération, à ce stade de l'année ! Tout doit se faire en ligne de commande dans le Terminal.**
2. Écrire une fonction `write_list_csv` qui prend en entrée une liste d'entiers et qui écrit pour chaque valeur : l'indice de cette valeur dans la liste, une virgule puis la valeur puis un retour à la ligne
   Le chemin absolu du fichier à écrire sera sera donné en entrée de la fonction.
3. Compléter les fonctions existantes pour qu'elles renvoient également, en plus des autres choses déjà renvoyées, la valeur du potentiel en sortie de l'opération.
4. Écrire une fonction récursive `successive_enqueue` qui permet d'enchaîner n opérations d'enfilage successives à partir d'une file initiale donnée en entrée, et qui permet d'accumuler les valeurs du potentiel pour ces n invocations dans une liste. On enfilera toujours le même caractère t. Le type de la fonction sera :

   ```
   successive_enqueue: int -> char queue -> int list -> char queue * int list <fun>
   ```

5. Écrire une fonction `potential` qui prend en entrée un entier n et renvoie la liste des potentiels obtenus pour n invocations successives de `enqueue` à partir d'une file de caractères vide.
6. Écrire une fonction récursive `successive_dequeue` qui permet d'enchaîner n opérations de défilage successives à partir d'une file initiale donnée en entrée, et qui permet d'accumuler les valeurs du potentiel pour ces n invocations dans une liste. On enfilera toujours le même caractère t. Le type de la fonction sera :

   ```
   successive_dequeue: int -> 'a queue -> int list -> 'a queue * int list <fun>
   ```

7. On représente une liste d'invocations « mixte » par une liste d'entiers. Si l'entier est positif, par exemple 5, il s'agit d'une série 5 enfilages (`enqueue`). Si l'entier est négatif, par exemple -7 il s'agit d'une série de 7 défilage (`dequeue`).
   Par exemple, la séquence d'invocations $[10; -5; 10; -10; 10; -10]$ signifie que l'on effectue 10 enfilages, puis 5 défilages, puis à nouveau 10 enfilages, puis 10 défilages...etc
   Écrire une fonction récursive `successive_ops` qui prend en entrée une telle séquence d'invocations et une liste initiale et renvoie la liste des potentiels obtenus à la sortie de chaque appel à `enqueue` ou `dequeue`.
8. Testez votre fonction sur plusieurs listes d'invocations comme celle décrite ci-dessus.
9. Utilisez la fonction `write_list_csv` pour améliorer votre fonction `potential` de sorte que l'évolution du potentiel soit écrite dans un fichier `complexite_amortie.csv`
10. Utilisez la bibliothèque `csv` de Python pour lire ce fichier et tracer l'évolution du potentiel. Vous pourrez aller voir la documentation en ligne ici : https://docs.python.org/fr/3/library/csv.html
11. Tracer les évolutions de potentiel pour plusieurs séquences différentes et observez.

Exercice 5 (Ligne de commande, redirection de flux et pipes).

1. Exécutez chacune des commandes suivantes (dans l'ordre) dans votre terminal, examinez le contenu des fichiers manipulés et en déduire ce qui se passe.

```
ls -R > file1
less file1
cat file1
cat file1 | less
ps aux
ps aux | grep -v root
ps aux | grep root
ps aux > file2
cat file2 | grep -v root
```

2. A quoi servent les opérateurs $>$, $<$ et $|$?
3. Tapez, dans le répertoire du TP d'aujourd'hui, la commande

```
ls ./*~
```

   Qu'obtenez vous ?
4. Tapez une commande permettant de lister tous les fichiers ayant l'extension `.c` à partir de la racine de votre répertoire utilisateur.

Le caractère $*$ est un caractère joker qui signifie *n'importe quelle chaîne de caractères*. L'argument `./*~` est ce que l'on appelle une `expression régulière`. Elles permettent de filtrer des chaînes de caractères selon un motif, un peu comme pour les motifs de déconstruction en OCaml. Elles sont très puissantes et sont abondamment utilisées dès lors que l'on écrit des commandes Linux.