TP n°6 - Correction

Entrées-sorties, exceptions et compilation séparée

Dans ce TP, on va écrire plusieurs programmes
- un programme qui lit des flottants dans le fichier `entree.lst` et écrit leur somme dans le fichier `sortie.txt`
- une variante qui écrit le produit de leurs successeurs au lieu de leur somme
- une variante qui lit les flottants depuis le clavier et affiche le résultat à l'écran
- une variante qui permet à l'utilisateur de choisir les noms des fichiers

ceci en minimisant au plus la duplication de code et les opérations de compilation

Attention : dans ce TP, on demande de *respecter à la lettre* les noms des fonctions ainsi que les noms des fichiers source

1 Préliminaire

Exercice 1 Créer le fichier `calcul.ml` et y écrire une fonction

```
calculer : float list -> float
```

qui, étant donnée une liste $[l_1\ l_2\ \dots\ l_n]$ de flottants, calcule leur somme. On renverra 0 si la liste est vide.

On pourra utiliser la fonction

```
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

qui, étant données une opération binaire $\perp$, une liste $l = [l_1\ l_2\ \dots\ l_n]$, calcule, pour tout a

$$\texttt{List.fold\_left}\ \perp\ a\ l = a \perp l_1 \perp l_2 \dots \perp l_n$$

$$\texttt{List.fold\_left}\ \perp\ a\ [] = a$$

Correction :

```
let calculer = List.fold_left ( +. ) 0.
```

2 Entrée-sortie depuis un fichier

Exercice 2 Créer le fichier `entreesortie.ml` et y écrire un programme qui lit indéfiniment des flottants dans un fichier `bidon.lst` et, chaque fois qu'un flottant f est lu, calcule et affiche $f+1$.

On ne demande pas que ce programme termine : on pourra supposer que `bidon.lst` est infini, cependant **on n'utilisera pas de boucle `while`** : on écrira plutôt une fonction auxiliaire récursive.

On pourra utiliser les fonctions

– `open_in : string -> in_channel` ouvre en ecture seu e e chier dont e nom est passé en argument, et renvoie un cana d entrée correspondant
– `input_line : in_channel -> string` it une igne dans e cana d entrée passé en argument
– `float_of_string : string -> float` véri e si a cha ne passée en argument est un ottant, et e renvoie dans ce cas
– `print_float : float -> unit` a che un ottant à écran

Tester ce programme en créant un chier `bidon.lst` et en y entrant un ottant par igne Que se passe-t-i
– orsque e programme arrive à a n du chier
– si une des ignes ne contient pas de ottant

Le programme peut-i terminer autrement

Correction :

```
let entree = open_in "bidon.lst" in
 let rec aux () =
  print_float (float_of_string (input_line entree) +. 1.);
  aux ()
 in aux ()
```

À a n du chier, exception `End_of_file` est ancée
Si un non- ottant est u une exception `Invalid_argument("float_of_string")` est ancée
Ce programme ne peut pas terminer autrement qu en ançant une de ces deux exceptions

Exercice 3 [Entrée depuis un chier] Modi er e chier `entreesortie.ml` pour y écrire une fonction `lire : in_channel -> float list` qui, étant donné un cana d entrée supposé déjà ouvert, y it des ottants jusqu à a n du chier d entrée, ève une exception si une des ignes ues n est pas un ottant, et sinon, renvoie a iste de tous es ottants us

On demande que es ottants de a iste renvoyée soient dans ordre où i s ont été us dans e chier

Remarque : on ne fermera pas e cana d entrée dans a fonction `lire` C est dans e programme testant cette fonction qu i faut e faire, en uti isant `close_in : in_channel -> unit`

Correction : Une version naïve

```
let rec lire entree =
 try
  let f = float_of_string (input_line entree)
  in f :: lire entree
 with End_of_file -> []
```

Cette fonction [illegible]

```
let lire entree =
 let rec aux accu =
  try
   let f = float_of_string (input_line entree)
   in aux (f :: accu)
  with End_of_file -> List.rev accu
 in aux []
```

Cette fonction accumule les éléments en les empilant dans la liste. Il est facile de voir qu'ils se retrouvent dans l'accumulateur `accu` *dans l'ordre inverse de leur lecture*, ce qui justifie, en fin de chier, de retourner l'accumulateur avant de le renvoyer.

Si on veut être précis sur la capture des exceptions, on peut utiliser un type `option`.

```
let lire entree =
 let rec aux accu =
  match
   try
    Some (input_line entree)
   with End_of_file -> None
  with
   | Some l -> aux (float_of_string l :: accu)
   | None   -> List.rev accu
 in aux []
```

3 Compilation séparée

Exercice 4

1. Mettre en commentaire tous les codes de test dans les fichiers `calcul.ml` et `entreesortie.ml`, puis compiler le fichier `calcul.ml` en utilisant, dans un terminal, la commande `ocamlc -c calcul.ml` (on peut aussi utiliser la commande Tuareg > Compile du mode Tuareg d'`emacs`).

   Puis, *sans modifier le fichier* `calcul.ml`, ni recopier son contenu, écrire dans le fichier `entreesortie.ml` à la suite de `lire`, une fonction

   `dialoguer : in_channel -> out_channel -> unit`

   qui, étant donnés un canal d'entrée et un canal de sortie supposés déjà ouverts, utilise les fonctions définies précédemment pour lire dans le canal d'entrée les flottants et pour calculer leur somme, et écrit dans le canal de sortie la somme ainsi obtenue.

   On utilisera la fonction `output_string : out_channel -> string -> unit` pour écrire dans le fichier de sortie. Il faudra donc aussi utiliser `string_of_float : float -> string`.

   On ne fermera pas les canaux d'entrée ni de sortie.

   Correction :

   ```
   let dialoguer entree sortie =
    output_string sortie (string_of_float (Calcul.calculer (lire entree)))
   ```

3. Puis compiler le fichier `entreesortie.ml` ainsi obtenu et écrire un fichier `programme.ml` qui utilise la fonction `dialoguer` pour lire dans le fichier `entree.lst` et écrire dans le fichier `sortie.txt`.

   Pour `programme.ml` on utilisera les fonctions suivantes :
   - `open_out : string -> out_channel` ouvre en écriture seule le fichier dont le nom est passé en argument, et renvoie le canal de sortie correspondant.

– `close_out : out_channel -> unit` ferme le fichier ouvert en écriture. Il est **important** de l'utiliser systématiquement car la plupart des systèmes d'exploitation n'écrivent pas le fichier sur disque tant qu'il n'est pas fermé.

Correction : Une version naïve

```
let entree = open_in  "entree.lst" ;;
let sortie = open_out "sortie.txt" ;;
Entreesortie.dialoguer entree sortie;;
close_in  entree;;
close_out sortie;;
```

Une version en une seule instruction

```
let entree = open_in  "entree.lst" in
let sortie = open_out "sortie.txt" in
 Entreesortie.dialoguer entree sortie;
 close_in  entree;
 close_out sortie
;;
```

On peut tester le programme ainsi écrit avant de le compiler, cependant il faudra veiller à

```
val calculer : float list -> float
```

Puis supprimer tous es chiers *.cm* ainsi que progsomme, et recompi er tout e programme étape par étape, en commençant par calcul.mli, et en produisant e programme progproduitsucc au ieu de progsomme

3 Puis copier e chier calcul.ml en calculproduitsucc.ml

Puis copier e chier calculsomme.ml en calcul.ml et recompi er calcul.ml seu ement

5 Sans recompi er es autres chiers, e ectuer étape de iaison pour produire e programme progsomme Que se passe-t-i

Correction : Ça marche

Exercice 7 Écrire un chier Makefile pour automatiser e processus de compi ation mais pas e remp acement du chier calcul.ml pour produire un programme prog Ce chier devra comporter une règ e pour chacun des chiers produits calcul.cmi, calcul.cmo, entreesortie.cmo, programme.cmo et prog

4 ti iser a commande ocamldep *.ml* pour véri er es dépendances entre modu es

Correction :

```
calcul.cmi: calcul.mli
ocamlc -c calcul.mli

calcul.cmo: calcul.cmi calcul.ml
ocamlc -c calcul.ml

entreesortie.cmo: calcul.cmi entreesortie.ml
ocamlc -c entreesortie.ml

programme.cmo: entreesortie.cmo programme.ml
ocamlc -c programme.ml

prog: calcul.cmo entreesortie.cmo programme.cmo
ocamlc -o prog calcul.cmo entreesortie.cmo programme.cmo
```

Copier e chier calculproduitsucc.ml sur calcul.ml puis entrer a commande make prog dans un termina Que se passe-t-i Tester e programme prog ainsi obtenu Recommencer avec calculsomme.ml

Correction : make ne recompi e que es chiers nécessaires, à savoir calcul.cmo Puis i e ectue étape de iaison

Exercice 8 Après en avoir fait une copie de sauvegarde, modi er e chier programme.ml pour ire es ottants depuis entrée standard c avier stdin et es a cher sur a sortie standard écran stdout Dans ce cas i ne faut pas fermer es canaux

Correction :

```
Entreesortie.dialoguer stdin stdout
;;
```

Recompi er avec make prog que se passe-t-i

Correction : make ne recompi e que es chiers nécessaires, à savoir programme.cmo Puis i e ectue étape de iaison

Cette fois-ci on remarque que a création d une interface pour entreesortie.ml n est pas nécessaire En e et, on suppose que ce modu e ne sera jamais modi é

Exercice 9 Reprendre le fichier `programme.ml` initial et le modifier pour permettre à l'utilisateur d'entrer au clavier le nom du fichier d'entrée et le nom du fichier de sortie. On utilisera la fonction `read_line : unit -> string` qui récupère une ligne tapée par l'utilisateur.

Pour chaque nom de fichier demandé, on affichera avec `print_endline` un message à l'écran.

Correction :

```
let nom_entree =
 print_endline "Nom du fichier d'entrée ?";
 read_line () in
let nom_sortie =
 print_endline "Nom du fichier de sortie ?";
 read_line () in
let entree = open_in  nom_entree in
let sortie = open_out nom_sortie in
 Entreesortie.dialoguer entree sortie;
 close_in  entree;
 close_out sortie
;;
```