Université Paris Diderot — Programmation Fonctionnelle
Licence — Année 2010-2011

TP n°6

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 $[l_1; l_2; \ldots; l_n]$ 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 $l = [l_1; l_2; \ldots; l_n]$, calcule, pour tout a :

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

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

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 lecture seule le fichier dont le nom est passé en argument, et renvoie un canal d'entrée correspondant
- `input_line : in_channel -> string` lit une ligne dans le canal d'entrée passé en argument

– `float_of_string : string -> float` vérifie si la chaîne passée en argument est un flot-

5. Puis effectuer l'étape de *liaison* produisant le programme final `progsomme` en utilisant la commande `ocamlc -o progsomme` suivie des fichiers `.cmo` produits. Enfin exécuter le programme `./progsomme` dans un terminal.

4 Recompilation séparée

On souhaite maintenant calculer le produit des successeurs au lieu de la somme, c'est-à-dire, étant donnée une liste $l = [l_1; l_2; \ldots; l_n]$, calculer $(1 + l_1) \times (1 + l_2) \times \ldots \times (1 + l_n)$ (renvoyer 1 si vide), et ce en effectuant le moins d'étapes de compilation possible.

Exercice 5 Copier le fichier `calcul.ml` en `calculsomme.ml`

Puis modifier le fichier `calcul.ml` pour que la fonction `calculer` calcule le produit des successeurs. On utilisera additionnellement la fonction `List.map`.

Recompiler `calcul.ml`.

Puis effectuer l'étape de liaison. Que se passe-t-il ?

Exercice 6

1. Écrire un fichier d'*interface* `calcul.mli` qui déclare la fonction `calculer` et son type, mais sans la définir.
2. Puis supprimer tous les fichiers `*.cm*` ainsi que `progsomme`, et recompiler tout le programme étape par étape, en commençant par `calcul.mli` , et en produisant le programme `progproduitsucc` au lieu de `progsomme`.
3. Puis copier le fichier `calcul.ml` en `calculproduitsucc.ml`
4. Puis copier le fichier `calculsomme.ml` en `calcul.ml` et recompiler `calcul.ml` seulement.
5. Sans recompiler les autres fichiers, effectuer l'étape de liaison pour produire le programme `progsomme`. Que se passe-t-il ?

Exercice 7 Écrire un fichier `Makefile` pour automatiser le processus de compilation (mais pas le remplacement du fichier `calcul.ml`) pour produire un programme `prog`. Ce fichier devra comporter une règle pour chacun des fichiers produits : `calcul.cmi`, `calcul.cmo`, `entreesortie.cmo`, `programme.cmo` et `prog`.

Utiliser la commande `ocamldep *.ml*` pour vérifier les dépendances entre modules.

Copier le fichier `calculproduitsucc.ml` sur `calcul.ml` puis entrer la commande `make prog` dans un terminal. Que se passe-t-il ? Tester le programme `prog` ainsi obtenu. Recommencer avec `calculsomme.ml`

Exercice 8 Après en avoir fait une copie de sauvegarde, modifier le fichier `programme.ml` pour lire les flottants depuis l'entrée standard clavier (`stdin`) et les afficher sur la sortie standard écran (`stdout`). Dans ce cas il ne faut pas fermer les canaux !

Recompiler avec `make prog` : que se passe-t-il ?

Exercice 9 Reprendre le fichier `programme.ml` initial et le modifier pour permette 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.