SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

MPIL(LU3IN008)
Année 2019/2020
Séance 3 de TD/TME

Objectifs

1. Programmation fonctionnelle.
2. Enregistrements.
3. Exceptions.
4. Écriture d'une boucle d'interaction.

Travaux Dirigés

Exercice 1 : Fonctions anonymes

Q.1.1 Que se passe-t-il lors de l'évaluation des lignes suivantes ?

```
let f () =
  let _ =
    print_string "foo\n"
  in
  1+1

let g =
  let _ =
    print_string "bar\n"
  in
  fun () -> 1+1

let () = f (); g (); f (); g ()
```

Q.1.2 Écrivez une fonction de type `unit -> int` qui renvoie un nouveau nombre entier unique à chaque appel, en un seul let (et donc sans passer par une référence globale).

Q.1.3 Sur le même modèle, écrivez deux fonctions de type `unit -> int` et `unit -> unit`, la première renvoyant un nouveau nombre entier unique à chaque appel et la seconde réinitialisant la génération.

Exercice 2 : Exceptions

Q.2.1 La fonction suivante est-elle récursive terminale ? Comment vérifier expérimentalement ?

```
let rec somme_tab_aux tab i acc =
  try
    match i with
      | i :: tl ->
        somme_tab_aux tab tl (acc + tab.(i))
      | [] -> acc
  with
    Invalid_argument "index␣out␣of␣bounds" ->
      Printf.fprintf stderr "out␣of␣bounds␣!" ;
      exit 1

let somme_tab tab i = somme_tab_aux tab i 0
```

Q.2.2 Quel problème voyez-vous dans la fonction suivante. Corrigez-le.

```
let rec lines fp =
  try
    input_line fp :: lines fp
  with End_of_file -> []
```

Q.2.3 Que se passe-t'il dans la fonction précédente si on attrape toutes les exceptions comme suit ?

```
let rec lines fp =
  try
    input_line fp :: lines fp
  with _ -> []
```

Exercice 3 : Jeu de rôles

Nous allons implanter un petit jeu de type RPG dans lequel le joueur a une classe, combat des monstres et ramasse leurs possessions lorsqu'il les a vaincus.

Q.3.1 Un personnage a une classe, (archer, barbare ou magicien), un nombre de points d'expérience, et un sac contenant divers objets en diverses quantités. Les objets peuvent être des pièces de monnaie, des cuisses de poulet ou des éponges.

Définissez les types en conséquence.

Q.3.2 Un monstre a une race (golem, sanglier ou moustique) et possède un objet dans sa poche que le personnage pourra ramasser triomphalement. On considérera des armées de moustiques en nombre variable comme un seul monstre.

Définissez les types en conséquence.

Q.3.3 Écrivez la fonction qui prend en paramètre un sac, un objet, et ajoute cet objet au sac.

Q.3.4 Écrivez la fonction affichant le contenu du sac. On demande un affichage sous la forme suivante :

```
2 eponges
1 poulet
2 pieces
```

Q.3.5 Écrivez une fonction générant aléatoirement un monstre. Vous pouvez utiliser la fonction `Random.int` qui prend un paramètre entier n et renvoie un nombre aléatoire entre 0 et n (non-inclu).

Pour obtenir des tirages pseudo-aléatoires différents entre deux lancement du programme, vous pouvez utiliser la fonction `Random.self_init ()` une fois au début du programme.

Q.3.6 Écrivez la fonction `frappe` calculant le nombre de points de vie enlevés à l'adversaire lors d'une attaque. Un barbare fait 10 points de dégâts, un archer 4 et un magicien 5. Cependant, avant de faire des dégâts, il doit réussir à toucher sa cible. Un barbare a 30% de chances de toucher, un magicien 50% et un archer 70%. Le personnage gagne 5% de bonus au toucher et un point de dégats par point d'expérience.

Q.3.7 Écrivez la fonction `frappe_monstre` sachant qu'un golem fait 4 points de dégâts, chaque moustique d'une armée 1/2 point et un sanglier 2 points.

Q.3.8 Définissez une exception `Mort` à lever quand le personnage meurt.

Q.3.9 Écrivez la fonction `combat` effectuant un combat entre un personnage p et un monstre m. Chacun des deux commence à 20 points de vie. Si le personnage ne meurt pas, il gagne un point d'expérience, ainsi que l'objet contenu dans la poche du monstre.

Q.3.10 Écrivez la fonction `malheureuse_rencontre`, qui génère un monstre aléatoirement et le fait s'affronter avec le personnage. La fonction devra aussi afficher le monstre généré afin d'admirer les exploits de votre valeureux guerrier.

Q.3.11 Écrivez la boucle principale du jeu, réalisant un nombre défini de combats avant d'afficher le butin final, à moins que le personnage ait failli dans sa quête, auquel cas vous devrez l'afficher.

Exemple de partie :

```
Vous tombez nez à nez avec une armée de 7 moustiques !
Vous tombez nez à nez avec un terrRRrrible golem !
Vous tombez nez à nez avec un terrRRrrible golem !
Vous êtes mort !.
```

ou bien

```
Vous tombez nez à nez avec une armée de 7 moustiques !
Vous tombez nez à nez avec un terrRRrrible golem !
Vous tombez nez à nez avec une armée de 6 moustiques !
Vous tombez nez à nez avec un terrRRrrible golem !
Vous tombez nez à nez avec un terrRRrrible golem !
Bravo, vous avez vaincu !
Voici votre butin :
2 eponges
1 poulet
2 pieces
```

Travaux sur Machines Encadrés

Exercice 4

Dans ce problème nous voulons réaliser un programme de gestion de dictionnaire. Pour ce faire, nous allons utiliser une représentation en arbre lexical pour stocker les mots.

Q.4.1 Un tel arbre, de type `arbre_lex`, est de la forme suivante : c'est une liste de noeuds où un noeud est du type `Lettre of char * bool * arbre_lex`.
Donnez la déclaration du type `arbre_lex` en OCaml.

La valeur booléenne du noeud permet d'identifier la fin d'un mot. Pour mieux comprendre cette représentation demandez un exemple à votre professeur.

Q.4.2 Écrivez la fonction `existe` qui teste si un mot (une chaîne de caractère) appartient à un dictionnaire de type `arbre_lex`.

Q.4.3 Écrivez la fonction `ajoute` qui prend un mot et un dictionnaire et retourne un nouveau dictionnaire contenant ce mot.

La fonction devra lever (`Deja_defini` *mot*) si celui-ci était déjà présent.

Q.4.4 Écrivez la fonction `construit` qui prend en entrée une liste de mots et renvoie le dictionnaire correspondant.

Q.4.5 Écrivez la fonction `liste_de_dict` qui réalise l'opération inverse de la précédente.

Q.4.6 Écrivez la fonction `affiche` qui prend en entrée un dictionnaire et affiche tous les mots de ce dernier.

Q.4.7 Réalisez un programme OCaml qui lit interactivement des commandes à effectuer sur un dictionnaire. Le programme maintiendra une référence contenant initialement le dictionaire vide, et qui peut être mise à jour avec les commandes `ajoute` *mot*, `existe` *mot*, `affiche` et `quitte`.

Note : vous pouvez écrire vous même la séparation de la commande et de ses arguments, ou bien vous pouvez utiliser la librairie d'expressions régulières d'OCaml `Str`.

Pour compiler si vous utiliser `Str`, vous devez utiliser :
`ocamlc str.cma -o prog prog.ml`
ou avec le compilateur optimisé
`ocamlopt str.cmxa -o prog prog.ml`

Pour le *toplevel*, il faut charger `str.cma` :
`#load "str.cma"`

Q.4.8 Augmentez le programme de la question précédente en permettant l'enregistrement du dictionnaire dans un ficher texte par les commandes `enregistre` *fichier* et `ouvre` *fichier*.

Exemple d'interaction :

```
Bienvenue !
> ajoute azerty
> ajoute azerto
> ajoute azerto
Erreur: deja defini.
> affiche
azerty
azerto
> existe azerty
oui
> existe 12
non
> achete pizza
Erreur: commande inconnue ou mal ecrite.
> quitte
```

Exercice 5 : Implémentation d'une file d'attente avec deux listes

Il est assez courant, en programmation fonctionnelle, d'implémenter une file d'attente (ordre First in First Out – premier entré, premier sorti) à l'aide de deux structures de piles (Last In First Out – dernier entré, premier sorti). Dans cet exercice, nous allons donc implémenter une file à l'aide de deux listes.

Dans un fichier `queue.ml`, on définira une file d'attente avec le type suivant :

`type 'a queue = { debut:'a list; fin:'a list}`

- Les éléments seront ajoutés à la file en les mettant en tête de la liste `fin`
- Les éléments seront ôtés de la file en retirant l'élément en tête de la liste `debut`
- Dès que la liste `debut` sera vide, alors la liste `fin` retournée prendra sa place, et la nouvelle liste `fin` deviendra la liste vide.

Q.5.1 Définissez la fonction `create : unit -> 'a queue`, qui crée une file d'attente vide.

Q.5.2 Définissez la fonction `push : 'a -> 'a queue -> 'a queue`, qui ajoute un élément dans une file d'attente, et retourne la file d'attente résultante.

Q.5.3 Définissez la fonction `pop : 'a queue -> ('a * 'a queue)`, qui retire l'élément en tête d'une file d'attente, et le retourne avec la nouvelle file. Si la file est vide, alors l'exception `Empty_queue` sera levée.

Q.5.4 Définissez la fonction `to_list : 'a queue -> 'a list`, qui convertit notre structure de file d'attente en liste (les éléments ajoutés le plus récemment seront situés à la fin de la liste résultante, tandis que les plus vieux seront en tête).

Q.5.5 Testez votre implémentation.