université PARIS-SACLAY

semaine du 14 décembre 2020

Introduction à la programmation fonctionnelle
L2 Informatique

TP n° 7

Consignes les exercices ou questions marqués d'un ⋆ devront être rédigés sur papier (afin de se préparer aux épreuves écrites de l'examen). En particulier, il est recommandé d'être dans les même conditions qu'en examen : pas de document ni de calculatrice. Tous les TPs se font sous Linux.

0 Prise en main de l'environnement

Pour faciliter les TPs en mode distanciel, nous utiliseront le serveur Jupiter hébergé par le LAL (Laboartoire Accélérateur Linéaire). Dans cet environnement, vous avez accès, via un navigateur Web à :

- un espace de stockage
- un terminal Unix
- un éditeur de texte avec la coloration syntaxique pour OCaml
- les commandes vues en cours : `ocamlc`, `ocaml`, ...

1. Se rendre sur `https://jupytercloud.lal.in2p3.fr/`
2. Cliquer sur se connecter
3. Utiliser l'authentification centralisée de Paris-Saclay (si vous êtes déjà connecté via eCampus, il est possible que l'authentification se fasse automatiquement)
4. Si vous êtes redirigé vers la page de garde : `https://jupytercloud.lal.in2p3.fr/`, c'est un bug connu. Effacez vos cookies ou ouvrez l'URL dans une fenêtre de navigation privée
5. Arrivé sur l'interface, choisissez « Start My Server » (bouton vert)

Vous vous retrouvez dans un navigateur de fichier. Vous pouvez créer des répertoires et des fichiers. Le bouton « New » à droite comporte plusieurs options dont :

- Create Terminal : ouvre un terminal dans un autre onglet de votre navigateur
- Create Texte File : ouvre un éditeur de texte

Dans l'éditeur de texte, si vous renommez le fichier en lui donnant l'extension « `.ml` » la coloration syntaxique pour OCaml est activée.

Il est **vivement recommandé** de télécharger vos fichiers après chaque exercice, pour en avoir une copie sur votre ordinateur personnel. Les sauvegardes sur cet espace de travail expérimental ne sont pas garanties, et c'est un espace distinct de vos répertoires personnels sur les machines de l'Université.

1 Le compte est bon

Le but de l'exercice est d'implémenter le simulateur du jeu « le compte est bon » dont on a fait la conception pendant le cours.

Dans ce jeu, un nombre (la cible) est tiré au hasard entre 100 et 999 (inclus). On tire ensuite au hasards 6 cartes parmi 24 possibles : — les vingt cartes 1, 1, 2, 2, . . ., 9, 9, 10, 10 — les quatre cartes 25, 50, 75 et 100 le but du jeu est de pouvoir calculer le nombre cible en utilisant les cartes au plus une fois chacune et en les combinant avec les quatre opérations : addition, soustraction, multiplication, division. On a le droit d'effectuer une soustraction uniquement si le résultat est positif et on a le droit d'effectuer une division uniquement si le résultat tombe juste (i.e. le reste dans la division euclidienne vaut 0).

On se concentre dans la première partie sur les fonctions auxiliaires puis dans la seconde sur l'algorithme de résolution. On complètera le fichier `lceb.ml` fourni sur la page du cours.

1.1 Fonctions sur les listes

1. Écrire une fonction `nb_aleatoire : unit -> int` ne prenant pas d'argument et renvoyant un entier dans l'intervalle 100 – 999. On rappelle que `Random.int n` renvoie un entier entre 0 inclus et `n` exclus.
2. Écrire une fonction `gen_cartes_list : unit -> int list` qui génère la liste des 24 cartes du jeu dans l'ordre `[1;1;2;2;...;10;10;25;50;75;100]`.

   Attention : Votre fonction ne doit pas alouer directement la liste finale comme une constante mais la calculer au moyen d'une fonction auxiliaire, récursive terminale.
3. Écrire une fonction `n_premiers : int -> 'a list -> 'a list` qui renvoie les `n` premiers éléments d'une liste (en préservant leur ordre). Si l'entier donné est trop grand, la fonction lève une exception avec `failwith`
4. Écrire une fonction `choix : int list -> int list` prenant en argument une liste de cartes telles que renvoyée par la fonction `gen_cartes_list` et qui renvoie une liste de 6 de ces cartes tirées au hasard. On poura utiliser l'algorithme suivant :
   - Associer à chaque carte un entier aléatoire
   - Trier la liste obtenue par ordre croissant
   - Ne garder que les 6 premiers éléments de la liste triée
   - Retirer l'entier aléatoire de chaque carte
5. Écrire une fonction `retire i j l` qui renvoie la liste `l` privée des éléments en position `i` et `j`. On suppose que `i` et `j` sont des positions valides et distinctes (vous n'avez pas à le vérifier).

1.2 Résolution

On rappelle la méthode de résolution récursive vue en cours. Étant donné une liste de `cartes` (i.e. d'entiers).

- Pour chaque paire de cartes a et b dans `cartes` :
  - pour chaque opération `op` dans $\{+, -, \times, \div\}$:
    - si $r = a\ op\ b$ est valide :
      - si $r =$ `cible` alors on a trouvé la solution
      - sinon rechercher récursivement sur la liste `cartes` privée de a et b dans laquelle on ajoute r
- Quand toutes les possibilité ont été explorées sans rien trouver, on un échec.

Le pseudo-code ci-dessus masque plein de détails. En particulier, on veut pouvoir s'arrêter dès qu'on a trouvé une solution, même en étant au sein d'une fonction récursive. De plus on doit se souvenir des opérations que l'on a effectuées pour pouvoir les réafficher ensuite. Enfin, il nous faut une façon de représenter les opérations que l'on veut faire. On introduit pour cela deux types :

```
1 type binop = Add | Sub | Mul | Div
2 ;;
3 type res = Res of (binop * int * int) * int
4       | Invalid
5 ;;
```

Le type `binop` représente l'une des quatres opérations. Le type `res` représente le résultat d'une opération entre deux cartes ou le fait que cette opération est invalide.

Par exemple, `Res((Mul, 4, 3), 12)` (attention aux parenthèses) représente le fait que $4 \times 3 = 12$. Le constructeur `Res` attend *deux* arguments : le premier est un triplet de l'opération et de ces deux opérandes et le second est le résultat.

1. Compléter les fonctions `add`, `mul`, `sub`, `div`. Elles sont toutes de type : `int -> int -> res`. L'addition et la multiplication renvoient toujours un résultat valide. Pour la soustraction `sub a b`, si `a = b` alors le résultat est invalide (il est inutile de calculer 0 dans le jeu le compte est bon). Sinon calculer le résultat de la soustraction entre `a` et `b` ou `b` et `a` (selon que `a` est plus grand ou plus petit que `b`).

   Procéder de façon similaire pour la division entre `a` et `b` ou `b` et `a` (selon que `a` est plus grand ou plus petit que `b`). Si le résultat tombe juste, le renvoyer sinon renvoyer `Invalid`.
2. La fonction `op a b` renvoie la liste de toutes les opérations valides entre deux entiers `a` et `b` sous la forme de paires semblables aux arguments du constructeur `Res`. Par exemple `op 3 4` renvoie :

```
[ ((Add, 3, 4), 7); ((Mul, 3, 4),12); ((Sub, 4, 3), 1) ]
```

En effet, la soustraction est valide si on considère `4−3` plutôt que `3−4` et la division n'est pas valide car `4÷3` n'est pas un entier. Donner le type de la fonction anonyme passée en argument à `List.fold_left`.

3. Compléter la fonction `recherche cible cartes`. En particulier, dans la fonction interne il faut :
   — retirer des cartes restantes les cartes en position `i` et `j`. On appelle cette nouvelle liste `n_cartes_res`.
   — calculer la liste `valid_ops` de toutes les opérations valides entre `a` et `b`.
   — pour chaque `(op, r)` dans `valid_ops` :
      — si `r = cible` lever l'exception `Trouve` avec la liste des opérations accumulées en argument
      — sinon ajouter `(op, r)` à la liste des opérations accumulées et rechercher récursivement la solution dans les cartes `r :: n_cartes_res`
4. Compléter la fonction `lceb` pour afficher la solution trouvée. On fera attention à l'ordre dans le quel sont accumulés les opérations. Un exemple d'affichage du programme est :

```
Nombre cible: 278
Cartes: 50 5 8 75 7 5
Solution:
 => 50 / 5 = 10
 => 10 + 8 = 18
 => 75 - 18 = 57
 => 57 * 5 = 285
 => 285 - 7 = 27
```