Programmation Fonctionnelle : TP1

Université de Tours

`Département informatique de Blois`

Fonctions, conditions et structures de données

*
* *

Première partie

Dans cette première partie, on s'intéresse au problème classique en algorithmique de la vérification de la validité d'une date donnée. Ce problème est un bon exemple de la puissance du paradigme fonctionnel car il met en jeu différentes fonctions simples opérant ensemble sur un problème plus complexe.

Principe de la décomposition de problèmes

On considère une fonction `dateValide` de paramètres d'entrée : un jour j, un mois m et une année a et correspondant à la spécification suivante :

$$\texttt{dateValide} : \begin{cases} \texttt{int} \times \texttt{int} \times \texttt{int} & \to \{true, false\} \\ j, m, a & \mapsto \begin{cases} true & \text{si le format de date } j/m/a \text{ est valide} \\ false & \text{sinon} \end{cases} \end{cases}$$

Par exemple, pour les applications de cette fonction aux arguments suivants, on doit avoir les réponses :

- `dateValide 18 11 1999 :- true`
- `dateValide 18 13 1999 :- false` (Le mois numéro 13 n'existe pas)
- `dateValide 31 11 1999 :- false` (31 n'est pas un jour valide pour le mois de Novembre)
- `dateValide 29 2 1996 :- true`
- `dateValide 29 2 1999 :- false` (Le 29 Février n'existe pas pour l'année 1999)

Description du problème

1. Décrire en français les différentes conditions permettant de vérifier qu'une date au format $j/m/a$ est valide.
2. Écrire la spécification et le code d'une fonction `estDansIntervalle n n0 n1` qui teste si $n \in [\![n_0, n_1]\!]$, c'est-à-dire si un nombre n appartient à l'intervalle de nombres entre n_0 et n_1. On supposera que $n_0 < n_1$.

   Exemple : `estDansIntervalle 3 (-1) 9 :- true`

   `estDansIntervalle 0 5 7 :- false`

3. Écrire la spécification et le code d'une fonction `nbJoursMois m a` qui retourne le nombre de jours du mois m pour l'année a.

   On pourra s'aider de la fonction `bissextile` : $\begin{cases} \texttt{int} & \to \{true, false\} \\ a & \mapsto \begin{cases} true & \text{si } a \text{ est bissextile} \\ false & \text{sinon} \end{cases} \end{cases}$

   ```
   let bissextile a = (a mod 400 = 0) || ((a mod 4 = 0) && (a mod 100 <> 0));;
   ```

   Exemple : `nbJoursMois 7 1995 :- 31`

   `nbJoursMois 2 1996 :- 29`

   `nbJoursMois 2 1999 :- 28`

4. Écrire le code de la fonction `dateValide` en vous aidant des fonctions précédentes.

À l'aide du type `date`

On souhaite ré-implémenter les fonctions précédentes mais à l'aide d'une structure de données définissant une date.

1. Créer un type `date` composé des trois nombres j, m et a et de la forme $j/m/a$.
2. Écrire la spécification et le code de la fonction `dateValide_type d` qui retourne vrai si et seulement si la date d possède un format valide.
3. Écrire la spécification et le code d'une fonction `lendemain d` qui retourne la date du lendemain de la date d passée en paramètre.

   Exemple : `lendemain {j=31; m=1; a=2019} :- {j = 1; m = 2; a = 2019}`

   `lendemain {j=6; m=6; a=2016} :- {j = 7; m = 6; a = 2016}`

   `lendemain {j=31; m=12; a=2018} :- {j = 1; m = 1; a = 2019}`

Deuxième partie

Cette deuxième partie consiste, maintenant que vous maîtrisez bien les fonctions et les structures de données (type), à implémenter un jeu de cartes classique.

Principes avancés sur les types

En Ocaml, il est tout à fait possible de faire référence à d'autres types lorsqu'on définit un type. Mieux encore, on peut faire une auto-référence à son propre type. Tout est possible ! `:)`
Par exemple, si l'on veut définir un type `peinture` qui ré-utilise et élargit le type `couleur` que l'on avait défini au TD2, on peut écrire :

```
type peinture =

      Teinte of couleur
    | Numero of int
    | Melange of peinture * peinture;;
```

On dit alors ici qu'une `peinture` $p \in \{\text{Teinte}, \text{Numero}, \text{Melange}\}$ où une Teinte est une (`couleur`), un Numéro est un entier (`int`) et un Mélange est un couple de deux peintures (`peinture * peinture`).

Modélisation d'un jeu de cartes

On rappelle qu'un jeu de cartes classique est composé de `cartes` de valeurs distinctes dont chacune est affiliée à une certaine `enseigne` $\{\diamondsuit, \heartsuit, \clubsuit, \spadesuit\}$.

1. Définir un type `enseigne` où une enseigne $e \in \{\text{Carreau}, \text{Coeur}, \text{Trefle}, \text{Pique}\}$.
2. Définir un type `carte` où une carte $c \in \{\text{As}, \text{Roi}, \text{Dame}, \text{Valet}, \text{Petite\_carte}, \text{Jocker}\}$ où $\forall c \neq \text{Jocker}$, c possède une enseigne et où la Petite_carte est représentée par une enseigne et une certaine valeur entière.

   Exemple :
   ```
   let c1 = As Coeur;;

   val carte :- c1 = As Coeur;;

   let c2 = Petite_carte (3, Pique);;

   val carte :- c2 = Petite_carte (3, Pique)
   ```

La belote (facultatif)

Pour illustrer le filtrage sur les types somme, on définit la valeur d'une carte à la *belote*.
Cette valeur dépend d'une enseigne particulière, celle de l'atout, choisie par les joueurs à chaque tour. Les cartes ont donc une certaine valeur initiale qui peut changer en fonction de si elle sont de l'enseigne de l'atout ou non :

- L'As vaut 11 points,
- Le Roi vaut 4 points,
- La Dame vaut 3 points,
- Le Valet vaut 2 d'ordinaire et 20 points s'il est d'atout,
- Le neuf compte d'ordinaire pour 0, mais vaut 14 points quand il est d'atout,
- Le dix vaut 10 points et les autres petites cartes valent 0.

1. Écrire la spécification et le code d'une fonction `valeur_carte atout` qui détermine la valeur des cartes d'un jeu en fonction d'une enseigne *atout* défini comme l'enseigne d'atout. On pourra utiliser la notion de type générique `_`.
2. Écrire la spécification et le code d'une fonction `combat c1 c2 atout` qui prend deux cartes c_1 et c_2 en entrée ainsi que l'enseigne d'atout *atout*, et qui redonne la carte de plus grande valeur en sortie.