2 Exercice : les nombres complexes

5 pts

sources : `Exo_et_Pb/Intro/Type_somme/Complexe/`

Un *nombre complexe* est représenté ici par son écriture algébrique $a + ib$ où a et b sont des nombres réels et i l'unité imaginaire telle que $i^2 = -1$. Le réel a est appelé *partie réelle* ; le réel b est appelé *partie imaginaire*.

On implémente l'ensemble des complexes grâce au type produit suivant :

```
type complexe = float*float
```

Q1. (0,5pt) Compléter la définition de la constante ci-dessous implémentant i , le nombre complexe dont la partie réelle est nulle et la partie imaginaire égale à 1. On prendra soin de respecter les contraintes de typage.

```
let i : complexe = 0., 1.
```

Un nombre complexe est *imaginaire pur* si et seulement si sa partie réelle est nulle.

Q2. (1pt) Implémenter la fonction :

Profil *estPur* : *complexe* $\to \mathbb{B}$
Sémantique : *estPur*(z) *est vrai si et seulement si z est un complexe imaginaire pur.*

Correction

```
let estPur (a,b : complexe) : bool =
  a = 0.
```

Bon, normalement, on ne teste pas l'égalité d'un réel à 0...

Q3. (0,5pt) Donner une expression OCAML permettant de vérifier que i est un imaginaire pur.

Correction

```
let _ = assert (estPur i)
```

Ok sans `assert`, ok avec `= true`

Le *conjugué* du nombre complexe $z = a + ib$ est le nombre complexe noté $\bar{z}$ et défini par $\bar{z} = a - ib$.

Q4. (1pt) Implémenter la fonction :

Profil *conj* : *complexe* $\to$ *complexe*
Sémantique : *conj*(z) *est* $\bar{z}$*, le complexe conjugué de z.*

Correction

```
let conj (a,b : complexe) : complexe =
  a, -b
```

Soit $z_1 = a_1 + ib_1$ et $z_2 = a_2 + ib_2$ deux nombres complexes. On définit le *produit* $z_1 z_2$ comme le nombre complexe de partie réelle $a_1 a_2 - b_1 b_2$ et de partie imaginiaire $a_1 b_2 + b_1 a_2$.

Q5. (1pt) Implémenter la fonction :

Profil *prod* : *complexe* $\to$ *complexe* $\to$ *complexe*
Sémantique : (*prod* z_1 z_2) *est le complexe* $z_1 z_2$.

Correction

```
let prod (a1,b1 : complexe) (a2,b2 : complexe) : complexe =
  a1*.a2 -. b1*.b2, a1*.b2 +. b1*.a2
```

Le *module* d'un nombre complexe z, noté $|z|$, est défini ici comme la racine carrée du produit de z avec son conjugué.

Q6. (1pt) Implémenter la fonction `taille` (on ne peut pas appeler cette fonction `module`, car c'est un mot-clé d'OCAML) :

Profil *taille* : *complexe* $\to \mathbb{R}$
Sémantique : (*taille* z) *est le réel* $\sqrt{z\bar{z}}$

On pourra utiliser la fonction prédéfinie OCAML `sqrt : float -> float` qui donne la racine carrée d'un réel. On veillera à ne pas ré-implémenter des fonctions déjà implémentées.

Correction

```
let taille (z : complexe) : complexe =
   sqrt (fst (prod z (conj z)));;
```

20 pts

3 Problème : jouons au tarot

Le tarot est un jeu de cartes. Dans ce jeu, il y a 78 cartes, décrites ci-dessous :

- L' *excuse*, une sorte de joker, qui sera modélisée par le constructeur constant *Excuse* :

  Profil *Excuse* : *carte*

- 56 cartes « normales » réparties en 14 cartes des quatre enseignes traditionnelles : pique (♤), cœur (♥), carreau (♦) et trèfle (♣). La différence avec un jeu usuel de 52 cartes est le cavalier, une figure s'intercalant entre la dame et le valet. Dans l'ordre croissant de force et de valeur, on trouve donc :
  - Les *petites*, de 1 à 10. Le 1 (aussi appelé l'as) est la plus petite carte contrairement à ce qui se pratique dans de nombreux autres jeux. Les petites seront modélisées par le constructeur avec argument *Petite* :

    Profil *Petite* : $\{1, \ldots, 10\} \rightarrow$ *normale*
  - Les *honneurs* : valet, cavalier, dame et roi qui seront modélisées par les constructeurs constants :

    Profil *Valet, Cavalier, Dame, Roi* : *normale*

- 21 cartes portant un numéro : ce sont les *atouts*. Le numéro indique la force de chaque atout. Le plus fort est le 21, le plus faible le 1. Ces cartes seront modélisées grâce au constructeur avec argument *Atout* :

  Profil *Atout* : $\{1, \ldots, 21\} \rightarrow$ *carte*

3.1 Modélisation

Le jeu de tarot sera modélisé grâce à la hiérarchie de types suivante :

```
type nat = int                      (* ≥ 0 *)

type normale =
  | Petite of nat (* restreint à 1, ..., 10 *)
  | Valet | Cavalier | Dame | Roi

type enseigne = Trefle | Pique | Carreau | Coeur

type carte =
  | Atout of nat (* restreint à 1, ..., 21 *)
  | Excuse
  | Nonatout of normale * enseigne
```

Une séquence de cartes, type `seqcarte`, est soit vide – constructeur constant *Vide* : *seqcarte* – soit obtenue en ajoutant une nouvelle carte à gauche d'une séquence de cartes existante – constructeur avec argument *A* : *carte* × *seqcarte* → *seqcarte*. D'où l'implémentation :

```
type seqcarte = Vide | A of carte * seqcarte
```


On appelle *main* la séquence des cartes qu'un joueur détient à un instant du jeu.

Q7. (1pt) Compléter la définition de la constante `cst_EX_MAIN` ci-dessous modélisant la main contenant les cartes : valet de pique, atout n° 9, excuse, roi de carreau, 9 de cœur, atout n° 21.

```
let cst_EX_MAIN : seqcarte = . . .
```

Correction

```
let cst_EX_MAIN: seqcarte = A( Nonatout(Valet,Pique),
                            A( Atout 9,
                            A( Excuse,
                            A( Nonatout(Roi,Carreau),
                            A( Nonatout(Petite 9,Coeur),
                            A( Atout 21,Vide))))))
```

3.2 Calcul des points gagnés

Un *pli* est la séquence des cartes gagnées à chaque tour de jeu, c'est-à-dire les cartes prises par chaque joueur[1].

À la fin de la partie, on compte les points contenus dans les plis. Chaque carte vaut un nombre précis de point, donné par la règle suivante (on omet ici de nombreuses règles subtiles).

- atout n° 1, excuse et atout n° 21 : 4.5 points,
- roi : 4.5 points,
- dame : 3.5 points,
- cavalier : 2.5 points,
- valet : 1.5 points,
- toute autre carte : 0.5 points.

Q8. (0,5pt) Combien de points vaut un pli qui ne contiendrait que `cst_EX_MAIN`, définie à la question précédente ?

Correction 16.0

1. ou groupe de joueurs, pour ceux qui connaissent les règles du tarot à cinq joueurs

Q9. (3pt) Spécifier puis réaliser une fonction *pointsCarte* qui retourne le nombre de points d'une carte donnée quelconque. Dans la spécification, on donnera deux exemples : le nombre de points d'un atout et celui d'une carte qui n'est ni un atout, ni l'excuse.

Correction 1,5 pour la spéc. ; 1,5 pour la réal.

Profil *pointsCarte* : $carte \to \mathbb{R}^{*+}$

Sémantique : *pointsCarte(c) est le nombre de points correspondant à la carte c.*

Exemples :

1. *pointsCarte* (*Atout* 1) = 4.5
2. *pointsCarte* (*Nonatout* (*Dame*, *Pique*)) = 3.5

Implémentation

```
let pointsCarte (c:carte) : float (* > 0 *) =
  match c with
  | Excuse | Atout 1 | Atout 21 | Nonatout(Roi,_) -> 4.5
  | Nonatout(Dame,_)                              -> 3.5
  | Nonatout(Cavalier,_)                          -> 2.5
  | Nonatout(Valet,_)                             -> 1.5
  | _                                             -> 0.5
```

Pour calculer le nombre de points d'un pli, on spécifie la fonction suivante :

Profil *pointsPli* : $seqcarte \to \mathbb{R}^{*+}$

Sémantique : *pointsPli(p) est le nombre de points total du pli p, séquence de cartes.*

Exemple : *pointsPli*(*cst_EX_MAIN*) = 16.0

Q10. (1,5pt) Implémenter *pointsPli*.

Correction

```
let rec pointsPli (s:seqcarte) : float =
  match s with
  | Vide    -> 0.
  | A(c,cs) -> pointsCarte(c) +. pointsPli(cs)
```

On sanctionne `int` au lieu de `float`, `0` au lieu de `0.`, + au lieu de `+.`

3.3 Les atouts

Q11. (2pt) Implémenter une fonction *nbAtout* : $seqcarte \to \mathbb{N}$; *nbAtout*(*s*) est le nombre d'atout de la séquence de cartes *s*.

Correction

```
let rec nbAtout (s:seqcarte) : nat =
  match s with
  | Vide    -> 0
  | A(c,cs) -> nbAtout(cs) + match c with Atout _ -> 1 | _ -> 0
```

Si un joueur possède plus de 10 atouts dans sa main initiale, on dit qu'il a une *poignée* ; il peut alors obtenir des points bonus. Plus précisément :

- Une *simple poignée* correspond à au moins 10 atouts et au plus 12 atouts ; le bonus est de 20 points.
- Une *double poignée* correspond à au moins 13 atouts et au plus 14 ; le bonus est de 30 points.
- Une *triple poignée* correspond à au moins 15 atouts ; le bonus est de 40 points.

Q12. (2pt) Implémenter la fonction *pointsBonus* : $seqcarte \to \mathbb{R}^+$, qui détermine en fonction de la main d'un joueur le nombre de points bonus attribués. On prendra soin de ne pas ré-implémenter une fonction déjà implémentée.

Correction

```
let pointsBonus (s:seqcarte) : float (* ≥ 0 *) =
  let nba = nbAtout s
  in if nba >= 15 then 40.
     else if nba >=13 then 30.
     else if nba >= 10 then 20.
     else 0.
```

On sanctionne si recodage de `nbAtout`

3.4 La triche

Afin de détecter les éventuels tricheurs, nous allons vérifier que les cartes ne sont jouées qu'une seule fois.

Q13. (2pt) Implémenter la fonction *estDans* : $carte \to seqcarte \to \mathbb{B}$; (*estDans c s*) est vrai si et seulement si la carte $c \in s$.

Dans cette question, on interdit la composition conditionnelle. Votre réponse ne devra donc pas contenir de `if then else`.

Correction

```
let rec estDans (c:carte) (s:seqcarte) : bool =
  match s with
  | Vide     -> false
  | A(c1,cs) -> c1 = c || estDans c cs
```

0 si `if then else`

Q14. (2pt) Implémenter la fonction *unique* : $seqcarte \to \mathbb{B}$; (*unique s*) est vrai si et seulement si *s* ne contient pas deux fois la même carte.

Correction

```
let rec unique (s:seqcarte) : bool  =
  match s with
  | Vide    -> true
  | A(c,cs) -> not (estDans c cs) && unique cs
```


VERSION ENSEIGNANT – NE PAS DIFFUSER

Q15. (3pt) Donner des équations récursives définissant la fonction *oter* : *carte* → *seqcarte* → *seqcarte* ; (*oter c s*) est la séquence de cartes obtenue en supprimant toutes les occurrences de *c* dans *s*. Par exemple :

- (*oter* Excuse cst_EX_MAIN) = A(Nonatout(Valet,Pique),
  A(Atout 9, A(Nonatout(Roi,Carreau),
  A(Nonatout(Petite 9,Coeur), A(Atout 21, Vide)))))
- (*oter* Atout(12) (A(Nonatout(Roi,Carreau),
  A(Atout 12, A(Excuse, A(Atout 12, Vide)))))) =
  A(Nonatout(Roi,Carreau), A(Excuse, Vide))

L'implémentation en OCAML n'est pas demandée, et sera considérée comme hors-sujet.

> **Correction**
>
> (1) $(oter\ c\ Vide) = Vide$ 0,5pt
>
> (2) $(oter\ c\ A(c, cs)) = (oter\ c\ cs)$ 1pt
>
> (3) $(oter\ c\ A(c_1, cs)) = A(c_1, oter\ c\ cs)$, avec $c \neq c_1$ 1,5pt (0,5 pour cond. de validité)

Q16. (3pt)

a) Définir une *mesure*.

> **Correction** 1 pt
> $mesure(c, s) \stackrel{def}{=} |s|$, où s – séquence de cartes – est vue comme un ensemble de A, et $|s|$ est le cardinal de s.

b) En déduire que $\forall c \in carte,\ \forall s \in seqcarte$, l'évaluation de (*oter c s*) termine.

> **Correction** 2 pts
>
> - *mesure* étant un cardinal, elle est à valeur dans $\mathbb{N}$.
> - $mesure(c,\ A(c_1, cs)) = |A(c_1, cs)| = 1 + |cs| > |cs| = mesure(cs)$, on a bien la stricte décroissance entre deux appels récursifs. Éventuellement $c = c_1$, cette preuve est donc valable pour les deux équations récursives.