UNIVERSITÉ GRENOBLE ALPES — UFR IM²AG — DÉPARTEMENT LICENCE SCIENCES ET TECHNOLOGIE

LICENCE SCIENCES & TECHNOLOGIES, 1re ANNÉE

INF201
ALGORITHMIQUE ET PROGRAMMATION FONCTIONNELLE

2021-2022

EXAMEN TERMINAL, 1re SESSION — mai 2021-2022

sources : Exams/21-22/S1/
dépôt local , dépôt distant { i21_examS1_21-22.tex
prologue_examS1_custom.tex , preambule_examS1_custom.tex
enonce.tex }

- Épreuve de 2,5 heures; document autorisé : une feuille a4 MANUSCRITE recto-verso; pour les étrangers uniquement : dictionnaire papier NON annoté. Tout appareil électronique INTERDIT.
- Le sujet comporte un problème et deux exercices . L'ensemble est noté sur 120 points. Le barème est donné à titre indicatif.
- Si vous êtes bloqué par une question, laissez de la place et passez à la suivante qui peut être moins difficile. Sur votre copie, il vous est demandé de RÉPONDRE AUX QUESTIONS **DANS L'ORDRE** DE L'ÉNONCÉ.

  Dans toute question, il est possible d'utiliser une fonction d'une question précédente sans l'avoir définie. En revanche, l'utilisation d'une fonction auxiliaire de votre invention n'est possible qu'après l'avoir spécifiée et réalisée.
- Prenez le temps de bien lire le sujet et de repérer les questions faciles.

Table des matières

1 Exercice : séquences folles

30 pts

Exo_et_Pb/Types_recursifs/Cons_gauche_droite//

Les listes natives d'OCaml sont construites par ajout à gauche. En TD/TP, une autre façon de manipuler des séquences d'éléments, par ajout à droite, a été examinée. Le but de cet exercice est d'en explorer une troisième : par ajout à gauche **ou** à droite.

On implémente ainsi l'ensemble des *séquences folles* :

$séqgd(\alpha) \overset{def}{=}$	$\{V\} \cup$	séquence vide
	$\{C_g(pr, fin) \ / \ pr \in \alpha, fin \in séqgd(\alpha)\} \cup$	ajout à gauche
	$\{C_d(déb, der) \ / \ déb \in séqgd(\alpha), der \in \alpha\}$	ajout à droite

Q1. (1.25pt) Donner le profil des constructeurs V, C_g et C_d.

> **Correction**
> $V : séqgd(\alpha)$ — 0.25pt
> $C_g : \alpha \times séqgd(\alpha) \to séqgd(\alpha)$ — 0.5pt
> $C_d : séqgd(\alpha) \times \alpha \to séqgd(\alpha)$ — 0.5pt

Q2. (1.25pt) Implémenter $séqgd(\alpha)$.

> **Correction**

```
type 'a seqgd = V                        (* séquence vide  0.25pt *)
              | Cg of 'a * 'a seqgd      (* ajout à gauche  0.5pt *)
              | Cd of 'a seqgd * 'a      (* ajout à droite  0.5pt *)
```

Q3. (1.5pt) Implémenter trois constantes de type $séqgd(\mathbb{Z})$ différentes s_a, s_b et s_c modélisant la même suite d'entiers : 1 4 2.

> **Correction**

```
let sa : int seqgd = Cg(1, Cg(4, Cg(2, V)))   (*  0.5pt *)
and sb : int seqgd = Cd(Cd(Cd(V,1), 4), 2)    (*  0.5pt *)
and sc : int seqgd = Cg(1, Cd(Cg(4, V), 2))   (*  0.5pt *)
```

1.1 Nombre d'éléments

SPÉCIFICATION 1	
PROFIL	$nbElt : séqgd(\alpha) \to \mathbb{N}$
SÉMANTIQUE	$(nbElt\ s)$ est le nombre d'éléments de s.

Q4. (0.75pt) Compléter les exemples suivants :
(i) $(nbElt\ s_a) =$ 3 0.25pt (ii) $(nbElt\ s_b) =$ 3 0.25pt (iii) $(nbElt\ s_c) =$ 3 0.25pt

Q5. (2.5pt) Donner des équations de récurrence nécessaires et suffisantes pour définir *nbElt*. L'implémentation en OCaml n'est pas demandée.

> Correction
>
> (1) $nbElt(V) = 0$ — 0.5pt
>
> (2) $nbElt(Cg(\_, reste)) = 1 + nbElt(reste)$ — 1pt
>
> (3) $nbElt(Cd(reste, \_)) = 1 + nbElt(reste)$ — 1pt

Q6. (2.5pt) Définir une mesure, puis montrer que $\forall s \in séqgd(\alpha)$, l'évaluation de $nbElt(s)$ termine.

> Correction
>
> $mesure(s) \stackrel{def}{=} |s|$, où s est vue comme un ensemble de constructeurs $\neq V$ — 0.5pt
>
> - *mesure* est bien à valeurs dans $\mathbb{N}$ puisque c'est un cardinal ; — 0.5pt
> - mesure membre gauche éq. (2) $=$ $mesure(Cg(\_, reste))$ $=$ $|Cg(\_, reste)|$ $=$ $1 + |reste| > |reste| = mesure(reste)$ $=$ mesure membre droit ; *mesure* est bien strictement décroissante entre deux appels récursifs. — 1pt
> - idem pour éq. (3) en remplaçant $Cg(\_, reste)$ par $Cd(reste, \_)$ — 0.5pt

1.2 Équivalence

La 3e question de cet exercice, vue ci-avant, montre que des séquences folles syntaxiquement différentes (s_a, s_b, s_c) peuvent modéliser la même suite (1 4 2). On souhaite donc définir une fonction de comparaison :

SPÉCIFICATION 2

PROFIL $equiv : séqgd(\alpha) \to séqgd(\alpha) \to \mathbb{B}$

SÉMANTIQUE $(equiv\ s_1\ s_2)$ est vrai si et seulement si s_1 et s_2 modélisent la même suite d'éléments.

Dans les exemples ci-dessous, $\wedge$ dénote la conjonction (le « et ») ; $\neg$ la négation (le « non ») :

EX. ET PROP.
(i) $\forall s \in séqgd(\alpha),\ (equiv\ s\ s)$
(ii) $\forall s_1 \in séqgd(\alpha),\ \forall s_2 \in séqgd(\alpha),\ (equiv\ s_1\ s_2) \implies (equiv\ s_2\ s_1)$
(iii) $(equiv\ s_a\ s_b) \wedge (equiv\ s_a\ s_c) \wedge (equiv\ s_b\ s_c)$
(iv) $\neg(equiv\ s_a\ Cg(2, Cd(Cg(4, V), 1)))$

Pour réaliser *equiv*, une fonction auxiliaire est nécessaire.

Q7. (1.75pt) Spécifier (profil, sémantique, exemples) une fonction *seqgdVlist* qui convertit une séquence folle en une liste native OCaml. On donnera trois exemples d'utilisation significatifs. La réalisation de *seqgdVlist* n'est pas demandée.

> Correction
>
> **SPÉCIFICATION 3**
>
> **PROFIL** $seqgdVlist : séqgd(\alpha) \to séq(\alpha)$ — 0.5pt
>
> **SÉMANTIQUE** $seqgdVlist(s)$ est la liste native correspondant à s. — 0.5pt
>
> **EX. ET PROP.**
> (i) $seqgdVlist(s_a) = [1 ; 4 ; 2]$ — 0.25pt
> (ii) $seqgdVlist(s_b) = [1 ; 4 ; 2]$ — 0.25pt
> (iii) $seqgdVlist(s_c) = [1 ; 4 ; 2]$ — 0.25pt

Q8. (1.5pt) En déduire une implémentation non récursive de *equiv*.

> Correction

```
let equiv (s1 : 'a seqgd) (s2 : 'a seqgd) : bool = (* 0.5pt *)
  (seqgdVlist s1) = (seqgdVlist s2)  (* 1pt *)
```

1.3 Conversion d'une liste native

On souhaite convertir une liste native OCaml en séquence folle ne comportant aucun ajout à gauche (constructeur C_g interdit dans tout ce paragraphe). Pour cela, une fonction auxiliaire est nécessaire :

SPÉCIFICATION 4

PROFIL $derdeb : séq(\alpha) \setminus \{[\ ]\} \to \alpha \times séq(\alpha)$

SÉMANTIQUE Posons $(der, déb) = derdeb(s)$; *der* est le dernier élément de s, *déb* est le début de s, c'est-à-dire s privé de son dernier élément.

Q9. (0.5pt) Expliquer pourquoi la séquence vide est exclue du domaine de *der_deb*.

> Correction
>
> Le dernier élément de la séquence vide n'est pas défini.

Q10. (1pt) Compléter les exemples suivants :
(i) $derdeb([1]) = \boxed{(1, [\ ])}$ 0.5pt (ii) $derdeb([3 ; 0 ; 1 ; 7]) = \boxed{(7, [3 ; 0 ; 1])}$ 0.5pt

Q11. (3pt) Implémenter récursivement *derdeb* par filtrage (construction `match`) et sans utiliser de fonctions auxiliaires.

> **Correction**
>
> ```
> let rec derdeb (s : 'a list (*non vide*)) : 'a * ('a list) =
>   match s with
>   | [pr] -> (pr,[])
>   | pr::fin -> let der,deb = derdeb fin in (der, pr::deb)
> [@@warning "-8"]
> ```
>
> On ne sanctionne pas l'absence de l'attribut de désactivation de l'avertissement.

Q12. (1.25pt) Lors de l'évaluation de la réponse à la question précédente par l'interpréteur OCaml, le message suivant est affiché :

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched: []
```

S'agit-il d'une erreur du programmeur ? Si oui, comment y remédier ? Si non, expliquer pourquoi OCaml affiche ce message, et ce que doit faire le programmeur.

> **Correction**
>
> Ce n'est pas une erreur du programmeur. 0.25pt
>
> OCaml affiche un avertissement lorsqu'un filtrage n'est pas exhaustif, ce qui est le cas ici puisque le motif `[]` n'est pas présent. 0.5pt
>
> Le programmeur peut ignorer ce message, en vertu du domaine de *derdeb* défini dans la spécif. 0.5pt

La conversion d'une liste en séquence folle est spécifiée ainsi :

SPÉCIFICATION 5

PROFIL	$listVseqgd$: $séq\,(\alpha) \to séqgd(\alpha)$
SÉMANTIQUE	$listVseqgd(s)$ est la séquence folle sans constructeur C_g correspondant à la liste native s.

Q13. (1.5pt) Compléter les exemples ci-dessous :

(i) $listVseqgd([2]) =$ $\boxed{Cd(V,2) \quad 0.5pt}$ (ii) $listVseqgd([1\,;\,2]) =$ $\boxed{Cd(Cd(V,1),2) \quad 1pt}$

Q14. (3pt) Implémenter *listVseqgd* en utilisant *derdeb*.

> **Correction**
>
> ```
> let rec listVseqgd (s : 'a list) : 'a seqgd =
>   if s = [] then
>     V
>   else
>     let der,deb = derdeb s in Cd(listVseqgd deb,der)
> ```


Q15. (3pt) Donner une autre implémentation, non récursive, de *listVseqgd* en utilisant un schéma d'ordre supérieur.

> **Correction**
>
> ```
> let listVseqgd : 'a list -> 'a seqgd =
>   List.fold_left (fun acc e -> Cd(acc,e)) V
> ```
>
> Ok si solution sans l'application partielle ; -0,5 si `right` au lieu de `left` ; -0,25 si inversion arguments.

On considère la fonction suivante :

SPÉCIFICATION 6

PROFIL	$mystere$: $séq\,(\mathbb{Z}) \to séqgd(\mathbb{Z})$
SÉMANTIQUE	?

RÉALISATION 6

```
let mystere (s : int list) : int seqgd =
  List.fold_right (fun e acc -> if e mod 2 = 0 then
                                  Cg(e,acc)
                                else
                                  Cd(acc,e))
    s V
```

Q16. (1.75pt) Quel est le résultat de l'évaluation de `mystere([2;1;3;4])` ?

> **Correction**
>
> `Cg (2, Cd (Cd (Cg (4, V), 3), 1))`
>
> On sanctionne de -0.75pt si oubli que `fold_right` traite les éléments de la droite vers la gauche (ce qui se traduira par un ordre inexact).

Q17. (2pt) Donner la sémantique de *mystere* en Français. Attention : on ne demande pas de traduire ou paraphraser le code, mais d'expliquer *ce que fait* cette fonction (et pas comment elle le fait).

> **Correction**
>
> $mystere(s)$ est la séquence folle où les pairs (resp. impairs) de s sont à ajoutés à gauche (resp. à droite).

total section = 30.0pt

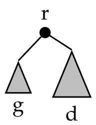
2 Exercice : séquence de niveaux

30 pts

Exo_et_Pb/Arbres/Seq_niveau/

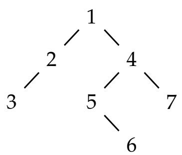
Le but de cet exercice est d'explorer la notion de niveau dans un arbre binaire.

Le niveau d'un nœud x dans un arbre peut être défini comme le nombre de nœuds sur la branche qui conduit de la racine de l'arbre jusqu'à x (inclus) ; la racine est donc de niveau 1.
On rappelle en outre la propriété :

Soit $p \in \mathbb{N}^*$. On considère un nœud x de niveau $p+1$ dans un arbre  . Alors x se situe soit dans g, soit dans d, et il est de niveau p dans le sous-arbre auquel il appartient (g ou d).

Q18. (1pt) Implémenter un type polymorphe *abin* (α) modélisant les arbres binaires.

> Correction
> ```
> type 'a abin = Av | Ab of 'a abin * 'a  * 'a abin
> ```
> -0,25 si `abin` au lieu de `'a abin`

On considère l'arbre suivant : $A =$ 

Q19. (2pt) Donner l'implémentation `cstA` de A.

Indication Procéder en définissant 3 ou 4 sous-arbres intermédiaires.

Il est conseillé d'utiliser les fonctions de construction $abS : \alpha \to abin\,(\alpha)$, $abUNg : abin\,(\alpha) \times \alpha \to abin\,(\alpha)$ et $abUNd : \alpha \times abin\,(\alpha) \to abin\,(\alpha)$, dont la réalisation n'est pas demandée .

> Correction
> ```
> let cstA : int abin =
>   let a23 = abUNg(abS 3, 2)
>   and a4567 =
>     let a56 = abUNd(5, abS 6) in Ab(a56, 4, abS 7)
>   in Ab(a23, 1, a4567)
> ```
> On ne sanctionne pas toute autre forme de (non-)imbrication des `let`, du moment que c'est syntaxiquement correct et que ça donne bien l'arbre demandé. Ok si syntaxe curryfiée. Ok aussi si utilisation uniquement des constructeurs.

Cette constante sera utilisée pour les exemples dans la suite de cet exercice.

2.1 Nombre de niveaux

Le nombre de niveaux d'un arbre est spécifié par :

SPÉCIFICATION 1	
PROFIL	$nbNiv : abin\,(\alpha) \to \mathbb{N}$
SÉMANTIQUE	$(nbNiv\ a)$ est la longueur de la plus grande branche de a.

> Aux enseignants
> $\equiv$ profondeur .

Q20. (0.5pt) Quel est le nombre de niveaux de l'arbre A ?

> Correction
> 4 ; n'importe quelle autre réponse $\Longrightarrow$ 0

Q21. (3pt) Donner des équations de récurrence nécessaires et suffisantes pour définir *nbvNiv*. L'implémentation en OCaml n'est pas demandée.

> Correction
>
> (1) $(nbNiv\ Av) = 0$ 1pt
>
> (2) $(nbNiv\ Ab(g,r,d)) = 1 + (max\ (nbNiv\ g)\ (nbNiv\ d))$ 2pt
>
> On accepte un appel non curryfié de `max` ; on ne sanctionne pas son inlining sous forme d'expression conditionnelle.

Q22. (4pt) Définir une mesure, puis montrer que $\forall a \in abin\,(\alpha)$, l'évaluation de $(nbNiv\ a)$ termine.

> Correction
>
> $(mesure\ a) \overset{def}{=} |a|$, où a est vu comme un ensemble (de `Ab`) (ou bien : où a est le nombre de nœuds) 1pt
>
> - *mesure* est bien à valeurs dans $\mathbb{N}$, puisque c'est un cardinal ; 1pt
> - mesure membre gauche éq. (2) $= (mesure\ Ab(g,r,d)) = |Ab(g,r,d)| = 1 + |g| + |d| > \begin{cases} |g| = mesure(g) \\ |d| = mesure(d) \end{cases}$ = mesures membre droit ;
>
>   *mesure* est bien strictement décroissante entre deux appels récursif. 2pt

2.2 Séquence de nœuds par niveau

La séquence des nœuds d'un niveau donné est spécifiée par :

SPÉCIFICATION 2

PROFIL $noeudsDeNiv : \mathbb{N}^* \to abin\,(\alpha) \to séq\,(\alpha)$

SÉMANTIQUE $(noeudsDeNiv\ n\ a)$ est la séquence des nœuds de niveau n dans a.

Q23. (1.25pt) Compléter les exemples suivants :

(i) $(noeudsDeNiv\ 1\ A) = \boxed{[\,1\,]}$

(ii) $(noeudsDeNiv\ 2\ A) = \boxed{[\,2\,;\,4\,]}$

(iii) $(noeudsDeNiv\ 3\ A) = \boxed{[\,3\,;5\,;7\,]}$

(iv) $(noeudsDeNiv\ 4\ A) = \boxed{[\,6\,]}$

(v) $(noeudsDeNiv\ 5\ A) = \boxed{[\,]}$

Q24. (4.25pt) En se basant sur la propriété rappelée en début d'exercice, implémenter *noeudsDeNiv*.

Correction

```
let rec noeudsDeNiv (n:nat (* ≠0 *)) (a : 'a abin) : 'a list =
  match a with
  | Av -> []
  | Ab(g,r,d) -> if n=1 then [r] else
      let n_1 = n-1 in
      (noeudsDeNiv n_1 g) @ (noeudsDeNiv n_1 d)
```

On compte 0,5 pour le profil ; on enlève 0,25 si `n-1` non factorisé.

2.3 Séquence des niveaux

La séquence des séquences de nœuds par niveau est spécifiée par :

SPÉCIFICATION 3

PROFIL $seqNiv : abin\,(\alpha) \to séq\,(séq\,(\alpha))$

SÉMANTIQUE Soit a un arbre non vide ; $(seqNiv\ a)$ est la séquence $[s_1\ ;\ \cdots\ ;\ s_n]$ $(n \geqslant 1)$ où s_i est la séquence (non vide) des nœuds de niveau i dans a. Si a est vide, il n'y a ni nœuds, ni niveau, et donc $(seqNiv\ a) = [\,]$.

Q25. (1pt) Compléter l'exemple suivant :
$(seqNiv\ A) = \boxed{[\,[1]\,;\,[2;4]\,;\,[3;5;7]\,;\,[6]\,]}$

2.3.1 Version 1 : réalisation récursive

Pour réaliser récursivement *seqNiv*, une fonction auxiliaire est nécessaire :

SPÉCIFICATION 4

PROFIL $zipseq : séq\,(séq\,(\alpha)) \to séq\,(séq\,(\alpha)) \to séq\,(séq\,(\alpha))$

SÉMANTIQUE
- $(zipseq\ s\ [\,]) = (zipseq\ [\,]\ s) = s$
- Soient n et $m \in \mathbb{N}^*$; $(zipseq\ [s_1\ ;\ \cdots\ ;\ s_n]\ [s'_1\ ;\ \cdots\ ;\ s'_m]) =$

$$\begin{cases} [s_1@s'_1\ ;\ \cdots\ ;\ s_n@s'_n\ ;\ s'_{n+1}\ ;\ \cdots\ ;\ s'_m], & \text{si } n < m \\ [s_1@s'_1\ ;\ \cdots\ ;\ s_n@s'_n], & \text{si } n = m \\ [s_1@s'_1\ ;\ \cdots\ ;\ s_m@s'_m\ ;\ s_{m+1}\ ;\ \cdots\ ;\ s_n], & \text{si } n > m \end{cases}$$

Q26. (2pt) Soient $s = [\,[2;1]\,;\,[0]\,;\,[3;1;2]\,]$ et $s' = [\,[3]\,;\,[4;2]\,;\,[\,]\,;\,[6]\,]$; compléter les exemples ci-dessous :

(i) $(zipseq\ s\ s') = \boxed{[\,[2;1;3]\,;\,[0;4;2]\,;\,[3;1;2]\,;\,[6]\,]}$

(ii) $(zipseq\ s'\ s) = \boxed{[\,[3;2;1]\,;\,[4;2;0]\,;\,[3;1;2]\,;\,[6]\,]}$

Q27. (4pt) Donner *au choix* des équations de récurrence nécessaires et suffisantes pour définir *zipseq*, ou bien une implémentation.

Correction

- Équations de récurrence :
  (1) $(zipseq\ [\,]\ s_2) = s_2$ 1pt
  (2) $(zipseq\ pr_1{::}fin_1\ [\,]) = pr_1{::}fin_1$ 1pt
  (3) $(zipseq\ pr_1{::}fin_1\ pr_2{::}fin_2) = (pr_1@pr_2){::}(zipseq\ fin_1\ fin_2)$ 2pt

ou bien :

```
let rec zipseq (s1 : 'a list list) (s2 : 'a list list) :
              'a list list =
  match s1 with
  | [] -> s2
  | pr1::fin1 -> match s2 with
    | [] -> s1
    | pr2::fin2 -> (pr1@pr2)::(zipseq fin1 fin2)
```

Q28. (3pt) En déduire *au choix* des équations de récurrence nécessaires et suffisantes pour définir *seqNiv*, ou bien une implémentation.

Correction

- Équations de récurrence :
  (1) $(seqNiv\ Av) = [\,]$ 1pt
  (2) $(seqNiv\ Ab(g,r,d)) = [r]{::}(zipseq\ (seqNiv\ g)\ (seqNiv\ d))$ 2pt

ou bien :

```
let rec seqNiv (a : 'a abin) : 'a list list =
  match a with
  | Av -> []
  | Ab(g,r,d) -> [r]::(zipseq (seqNiv g) (seqNiv d))
```

2.3.2 Version 2 : réalisation à l'ordre supérieur

Q29. (4pt) À l'aide d'un schéma d'ordre supérieur vu en cours/TD/TP, donner une implémentation non récursive de *seqNiv*.

Indication L'utilisation de la fonction suivante[1] sera nécessaire :

SPÉCIFICATION 5

PROFIL	*List.init* : $\mathbb{N} \to (\mathbb{N} \to \alpha) \to séq\,(\alpha)$
SÉMANTIQUE	$(List.init\ n\ f) = [f(0)\ ;\ \cdots\ ;\ f(n-1)]$
EX. ET PROP.	(i) $(List.init\ 3\ (e \mapsto e)) = [\,0\ ;\ 1\ ;\ 2\,]$

L'implémentation de `List.init` n'est pas demandée.

Correction

```
let seqNiv (a : 'a abin) : 'a list list =
  List.fold_right (fun e acc -> (noeudsDeNiv e a)::acc)
    (List.init (nbNiv a) ((+) 1)) []
```

total section = 30.0pt

[1] définie dans le module `List` de la librairie standard d'OCaml

3 Problème : le jeu de Takuzu

60 pts

Exo_et_Pb/Sequences/De_sequences/Takuzu/

3.1 Notations et indications

Il est rappelé que $seq^*(\alpha)$ désigne les séquences non vides de α : $seq^*(\alpha) = seq(\alpha) \setminus \{[\,]\}$, et que le cardinal (nombre de constructeurs) d'une séquence s est noté $|s|$.

On rappelle également que le type α étant quelconque, il peut être lui-même celui d'une séquence ; on manipule alors des séquences de séquences.

Les ensembles $\mathbb{N}$ et $\mathbb{N}^*$ seront utilisés, et implémentés grâce aux synonymes suivant :

```
type nat = int (* ⩾ 0 *)
type natpos = nat (* ≠ 0  *)
```

Dans les réponses que vous fournirez, sera considéré comme **hors-sujet** :

- quand il est indiqué d'utiliser une fonction particulière, toute solution n'utilisant pas cette fonction ;
- une implémentation OCaml, quand des équations sont demandées ;
- une implémentation récursive, quand une implémentation non récursive est demandée (et vice-versa).

Il est **interdit** d'utiliser les fonctions de la librairie standard d'OCaml autres que `List.map`, `List.fold` (`_left` ou `_right`), `List.hd` et `List.tl` (qui sont toutes les cinq autorisées).

3.2 Description du jeu

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
l_1		1		1		1		
l_2								
l_3								
l_4	0		0	0				
l_5		1				1		
l_6			0					
l_7				0				0
l_8	1					1	1	

a) phase initiale

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
l_1	0	1	0	1	0	1	0	1
l_2	1	0	1	0	1	0	0	1
l_3	1	0	1	1	0	0	1	0
l_4	0	1	0	0	1	1	0	1
l_5	0	1	1	0	0	1	1	0
l_6	1	0	0	1	1	0	0	1
l_7	0	1	1	0	1	0	1	0
l_8	1	0	0	1	0	1	1	0

b) phase finale

Le *Takuzu* (aussi appelé *Binairo*) est un jeu de réflexion à un joueur dont le but est de remplir une grille avec deux valeurs différentes, en général 0 ou 1. Initialement, la grille contient quelques valeurs placées aléatoirement. Pour terminer le jeu, le joueur doit la compléter jusqu'à ce qu'il n'y ait plus de cases vides.

La grille de jeu doit être de taille paire ; dans ce problème, elle sera de plus carrée. Le nombre de lignes est donc égal au nombre de colonnes, et ce nombre est un multiple de 2.

Ci-dessus un exemple de grille de Takuzu de taille 8, dans deux phases de jeu :

a) initialement, quand aucun coup n'a encore été joué ;

b) après le dernier coup.

Pour faciliter la lecture des grilles, les lignes et les colonnes ont été numérotées (de 1 à 8 sur cet exemple).

Les règles du jeu seront détaillées dans la section 3.4 ; il n'est pas nécessaire de les connaître pour l'instant.

Q30. (2pt) Définir un type énuméré appelé `valeur` modélisant les valeurs possibles dans une grille de Takuzu. On utilisera les constructeurs V, Z et U spécifiés ainsi :

SPÉCIFICATION 1

PROFIL V, Z, U : *valeur*

SÉMANTIQUE V représente une case vide, Z une case contenant 0 et U une case contenant 1.

Correction

```
type valeur = V | Z | U
```

Q31. (1pt) Implémenter une constante appelée `cstTAILLE` représentant la taille des grilles de Takuzu. On pourra par exemple prendre une taille de 4.

Correction

```
let cstTAILLE : natpos = 4
```

On ne sanctionne pas l'absence du type.

Q32. (4pt) Définir (spécification + réalisation) une fonction *chifVval* convertissant un entier en valeur de grille de Takuzu. À tout entier différent de 0 et de 1, on associera la case vide.

Correction 2pts chaque

SPÉCIFICATION 2

PROFIL *chifVval* : *natpos* → *valeur*

SÉMANTIQUE (*chifVval c*) est la valeur correspondant au chiffre c.

```
let chifVval (c:nat) : valeur =
  match c with 0 -> Z | 1 -> U | _ -> V
```


Q33. (1pt) Définir un type `takuzu` modélisant les grilles de Takuzu.

Indication une grille est modélisée comme une séquence de lignes.

Correction

```
type takuzu = valeur list list
(* nombre d'élt égal au nombre d'élmt de chaque sous-liste *)
```

On ne sanctionne pas l'abscence de la contrainte.

3.3 Fonctions utilitaires

On appellera *matrice* une séquence de séquences.

Dans cette section, on étudie quelques fonctions sur les matrices utiles à la gestion d'un jeu de Takuzu.

3.3.1 Élément (i, j) d'une matrice

On considère la fonction :

SPÉCIFICATION 3

PROFIL *ij_eme* : $\mathbb{N}^* \times \mathbb{N}^* \to seq^*(seq^*(\alpha)) \to \alpha$

SÉMANTIQUE (*ij_eme* (i, j) m) est le j-ème élément du i-ème élément de la matrice m. On pourra supposer que les contraintes suivantes sont vérifiées :

- $1 \leqslant i \leqslant |m|$;
- $1 \leqslant j \leqslant |s_i|$, où s_i est le i-ème élément de m.

EX. ET PROP. (i) (*ij_eme* $(2, 3)$ [[3 ;1 ;2] ;
[4 ;6 ;5] ;
[9 ;8 ;7]]) = 5

Q34. (4pt) Donner une implémentation <u>non</u> récursive de *ij_eme*. On utilisera la fonction :

SPÉCIFICATION 4

PROFIL *i_eme* : $\mathbb{N}^* \to seq^*(\alpha) \to \alpha$

SÉMANTIQUE (*i_eme i s*) est le i-ème élément de la séquence s. Le premier élément est obtenu avec $i = 1$. On pourra supposer que $1 \leqslant i \leqslant |s|$.

La réalisation de *i_eme* <u>n</u>'est <u>pas</u> demandée.

Correction

```
let ij_eme (i,j : natpos*natpos)
            (m : 'a list list (* toutes non vides *)) : 'a =
  i_eme j (i_eme i m)
```

3.3.2 Modification d'une matrice

On considère la fonction suivante de modification d'une séquence :

SPÉCIFICATION 5

PROFIL *modSeq* : $\mathbb{N}^* \to \alpha \to seq^*(\alpha) \to seq^*(\alpha)$

SÉMANTIQUE $(modSeq\ i\ x\ s)$ est la séquence obtenue en remplaçant dans s son i-ème élément par x. On pourra supposer que $1 \leqslant i \leqslant |s|$.

EX. ET PROP. (i) $(modSeq\ 2\ 0\ [3;1;2;1]) = [3;0;2;1]$

Q35. (6pt) Donner des équations de récurrence nécessaires et suffisantes pour définir *modSeq*. L'implémentation en OCaml n'est pas demandée.

Correction

(1) $(modSeq\ 1\ x\ pr::fin) = x::fin$ 2pt

(2) $(modSeq\ p{+}1\ x\ pr::fin) = pr::(modSeq\ p\ x\ fin)$, avec $p > 0$ 4pt

On sanctionne l'absence de $p > 0$ (-1pt), sans quoi l'équation est fausse.

Q36. (4pt) Définir une mesure, puis montrer que $\forall i \in \mathbb{N}^*,\ \forall x \in \alpha,\ \forall s \in séq\,(\alpha)$, l'évaluation de $(modSeq\ i\ x\ s)$ termine.

Correction

$(mesure\ i\ x\ s) \stackrel{def}{=} i$ 1pt

- *mesure* est bien à valeurs dans $\mathbb{N}$, d'après la spécif ; 1pt
- mesure membre gauche éq. (2) $= (mesure\ p{+}1\ x\ s) = p{+}1 > p = (mesure\ p\ x\ s) =$ mesure membre droit ; *mesure* est bien strictement décroissante entre deux appels récursifs. 2pt

Autre solution : $(mesure\ i\ x\ s) \stackrel{def}{=} |s|$

On considère la fonction suivante de modification d'une matrice :

SPÉCIFICATION 6

PROFIL *modMat* : $\mathbb{N}^* \times \mathbb{N}^* \to \alpha \to seq^*(seq^*(\alpha)) \to seq^*(seq^*(\alpha))$

SÉMANTIQUE $(modMat\ (i,j)\ x\ ss)$ est la matrice obtenue en remplaçant le j-ème élément du i-ème élément de ss par x. On pourra supposer que :

- $1 \leqslant i \leqslant |m|$;
- $1 \leqslant j \leqslant |s_i|$, où s_i est le i-ème élément de m.

Q37. (5pt) Déduire des questions précédentes une implémentation non récursive de *modMat*. On veillera à ne pas réécrire du code déjà écrit.

Correction

```
let modMat (i,j : natpos*natpos) (x:'a)
            (ss : 'a list list (* toutes non vides *)) :
            'a list list =
    modSeq i (modSeq j x (i_eme i ss)) ss
```

3.3.3 Colonne d'une matrice

On considère la fonction suivante d'extraction de colonne de matrice :

SPÉCIFICATION 7

PROFIL *colonne* : $\mathbb{N}^* \to seq(seq^*(\alpha)) \to seq(\alpha)$

SÉMANTIQUE $(colonne\ j\ m)$ est la j-ème colonne de la matrice m.

EX. ET PROP. (i) $(colonne\ 2\ [\ [3;1;2];\ [4;6;5];\ [9;8;7]\ ]) = [1;6;8]$

Q38. (4pt) En utilisant un schéma d'ordre supérieur, donner une implémentation non récursive de *colonne*.

Correction

```
let colonne (j:natpos) : 'a list list -> 'a list =
  map (fun e -> i_eme j e)
```

On accepte la solution sans application partielle. On accepte aussi une solution basée sur `fold`, par exemple :

```
let colonne (j:natpos) : 'a list list -> 'a list =
  fold_left (fun accu e -> accu @ [i_eme j e]) []
```

3.3.4 Matrice monotone

On considère la fonction suivante de création de séquence monotone :

SPÉCIFICATION 8

PROFIL *seqMono* : $\mathbb{N} \to \alpha \to seq(\alpha)$

SÉMANTIQUE $(seqMono\ lg\ x)$ fabrique la séquence de longueur lg contenant – si elle n'est pas vide – uniquement la valeur x.

EX. ET PROP. (i) $(seqMono\ 3\ 0) = [0;0;0]$

Q39. (4pt) Donner une implémentation récursive de *seqMono*

Correction

```
let rec seqMono (lg:nat) (x:'a) : 'a list =
  if lg = 0 then
    []
  else (* lg = p+1 *)
    x::(seqMono (lg-1) x)
```

On considère la fonction suivante de création de matrice monotone :

SPÉCIFICATION 9

PROFIL *matMono* : $\mathbb{N} \to \mathbb{N} \to \alpha \to seq(seq(\alpha))$

SÉMANTIQUE (*matMono* l c x) fabrique la séquence de l séquences monotones contenant chacune – quand elles ne sont pas vides – c fois la valeur x.

EX. ET PROP. (i) (*matMono* 2 3 0) = [[0;0;0] ; [0;0;0]]
(ii) (*matMono* 3 2 0) = [[0;0] ; [0;0]; [0;0]]

Q40. (6pt) Donner une réalisation de *matMono* sous forme d'équations de récurrence nécessaires et suffisantes, suivie d'une implémentation.

Correction

(1) $(matMono\ 0\ c\ x) = [\,]$ 1pt

(2) $(matMono\ p{+}1\ c\ x) = (seqMono\ c\ x) :: (matMono\ p\ c\ x)$ 2pt

```
let rec matMono (l:nat) (c:nat) (x:'a) : 'a list list =
  if l = 0 then
    []
  else (* l = p+1 *)
    (seqMono c x) :: (matMono (l-1) c x)
```

Q41. (3pt) Donner une autre implémentation, <u>non</u> récursive, de *matMono* <u>sans</u> utiliser de schéma d'ordre supérieur.

Correction

```
let matMono (l:nat) (c:nat) (x:'a) : 'a list list =
  seqMono l (seqMono c x)
```

3.4 Application des règles

Les règles du Takuzu sont :

« *MoitMoit* » chaque ligne et chaque colonne doit contenir autant de 0 que de 1

« *Max2consec* » il ne peut pas y avoir plus de deux valeurs identiques contigües (c'est-à-dire qui se suivent sur une même ligne ou une même colonne)

« *Unicit* » les lignes et les colonnes doivent être uniques.

La grille b) de l'exemple de la section 3.2 respecte les trois règles énoncées ci-dessus. Voici trois contre-exemples où les règles ne seraient pas respectées :

- Placer un 0 en c_7 de la grille

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
0	1	0	0	1	0		1

violerait la règle *MoitMoit*.

- Placer un 0 en c_4 de la grille

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
1	0	0		0		1	

ou en l_4 de la grille :

l_1	...	1	...
l_2	...	0	...
l_3	...	0	...
l_4	...		...
l_5	...	1	...
l_6	...	0	...
l_7	...	1	...
l_8	...	0	...

violerait la règle *Max2consec*.

- Obtenir la grille

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
0	1	0	0	1	0	1	1
0	1	0	0	1	0	1	1

violerait la règle *Unicit*.

Q42. (1pt) Compléter la grille :

	c_1	c_2	c_3	c_4
l_1	0			
l_2				0
l_3		1	0	
l_4	1	0		0

Correction

	c_1	c_2	c_3	c_4
l_1	0	0	1	1
l_2	1	1	0	0
l_3	0	1	0	1
l_4	1	0	1	0

Dans la suite, on s'intéresse uniquement à la règle *Max2consec*.

Q43. (6pt) Compléter les équations ci-après de la fonction :

SPÉCIFICATION 10

PROFIL $auMoins3ContEg : \alpha \to seq(\alpha) \to \mathbb{B}$

SÉMANTIQUE $(auMoins3ContEg\ v\ s)$ est vrai si et seulement si la séquence s comprend au moins 3 éléments contigus égaux à v.

Ci-dessous, on abrège l'identificateur *auMoins3ContEg* en *aM3CE*.

Correction

0.5pt si éq $\leqslant$ 3b, 1pt sinon

(1) $(aM3CE\ v\ [\,]) = \boxed{faux}$

(2) Soit $pr \neq v$, $(aM3CE\ v\ pr :: fin) = \boxed{(aM3CE\ v\ fin)}$

(3) $(aM3CE\ v\ v :: fin)$ se décompose en trois cas, selon la valeur de fin :

(3a) $(aM3CE\ v\ v :: [\,]) = \boxed{faux}$

(3b) Soit $pr_2 \neq v$, $(aM3CE\ v\ v :: pr_2 :: fin_2) = \boxed{(aM3CE\ v\ fin_2)}$

(3c) $\boxed{(aM3CE\ v\ v :: v :: fin_2)}$: 3 cas, selon la valeur de fin_2 :

(3c1) $\boxed{(aM3CE\ v\ v :: v :: [\,]) = faux}$

(3c2) $\boxed{\text{Soit } pr_3 \neq v,\ (aM3CE\ v\ v :: v :: pr_3 :: fin_3) = (aM3CE\ v\ fin_3)}$

(3c3) $\boxed{(aM3CE\ v\ v :: v :: v :: fin_3) = vrai}$

Q44. (4pt) Définir une mesure, puis montrer que $\forall v \in \alpha$, $\forall s \in séq\,(\alpha)$, l'évaluation de $(auMoins3ContEg\ v\ s)$ termine.

Correction

$(mesure\ v\ s) \overset{def}{=} |s|$. 1pt

- *mesure* est bien à valeurs dans $\mathbb{N}$, car c'est un cardinal ; 0pt (déjà évalué)
- mesure membre gauche éq. (2) $= (mesure\ v\ pr :: fin) = |pr :: fin| = 1 + |fin| > |fin| = (mesure\ v\ fin) =$ mesure membre droit ; 1pt
- mesure membre gauche éq. (3b) $= (mesure\ v\ v :: pr_2 :: fin_2) = |v :: pr_2 :: fin_2| = 2 + |fin_2| > |fin_2| = (mesure\ v\ fin_2) =$ mesure membre droit ; 1pt
- mesure membre gauche éq. (3c2) $= (mesure\ v\ v :: v :: pr_3 :: fin_3) = |v :: v :: pr_3 :: fin_3| = 3 + |fin_3| > |fin_3| = (mesure\ v\ fin_3) =$ mesure membre droit ; 1pt

mesure est bien strictement décroissante entre deux appels réc. 0pt (déjà évalué)

Q45. (5pt) Déduire des questions précédentes une implémentation non récursive de la fonction :

SPÉCIFICATION 11

PROFIL $reglMax2Consec : \mathbb{N}^* \times \mathbb{N}^* \to valeur \to takuzu \to \mathbb{B}$

SÉMANTIQUE $(reglMax2Consec\ (i,j)\ v\ tak)$ est vrai si et seulement si jouer le coup v sur la case (i,j) de *tak* respecte la règle *Max2Consec*.

On veillera à ne pas réécrire du code déjà écrit.

Correction

```
let contrMax2Consec (i,j : natpos*natpos) (v : valeur)
                    (tak : takuzu) : bool =
  let takjoue = modMat (i,j) v tak in
  not ((auMoins3ContEg v (i_eme i takjoue))
       || (auMoins3ContEg v (colonne j takjoue)))
```

3.5 Initialisation d'une grille de Takuzu (bonus)

Un moyen simple d'initialiser n cases d'une grille de Takuzu consiste à générer aléatoirement n couples distincts de coordonnées (i,j) (entiers compris entre 1 et `cstTAILLE`) désignant n cases vides, qui seront initialisées avec la valeur correspondant à $(i+j)$ *modulo* 2 ; cette valeur garantit le respect des règles *MoitMoit* et *Max2consec*, mais pas forcément celui de *Unicit* si n est trop proche de `cstTAILLE`[2] (revoir le paragraphe 3.4 pour le détail des règles).

On cherche à créer une grile partiellement remplie de cette manière, en partant d'une grille vide :

SPÉCIFICATION 12

PROFIL $initTakuzu : \mathbb{N} \to takuzu$

SÉMANTIQUE (*initTakuzu* n) fabrique une grille de Takuzu de taille `cstTaille` contenant n valeurs placées aléatoirement. On pourra supposer que $n \ll$ `cstTAILLE`^{2} de façon à ce que la règle *Unicit* soit automatiquement vérifiée.

La génération aléatoire d'entiers sera assurée par la fonction suivante de la librairie standard d'OCaml :

SPÉCIFICATION 13

PROFIL *Random.int* : $\mathbb{N}^* \to \mathbb{N}$

SÉMANTIQUE (*Random.int* x) génère aléatoirement un entier $\geqslant 0$ et $< x$.

Q46. (bonus) En utilisant les questions précédentes, implémenter *initTakuzu* récursivement.

Correction

```
let rec initTakuzu (n:nat) : takuzu =
  if n = 0 then
    matMono cstTAILLE cstTAILLE V
  else (* n > 0 *)
    let i = 1 + (Random.int cstTAILLE)
    and j = 1 + (Random.int cstTAILLE)
    and takn_1 = initTakuzu (n-1) in
    if ij_eme (i,j) takn_1 = V then
      modMat (i,j) (chifVval ((i+j) mod 2)) takn_1
    else
      initTakuzu n
```

8pts en bonus

total section = 60.0pt