UNIVERSITÉ GRENOBLE ALPES

UFR IM²AG

DÉPARTEMENT LICENCE SCIENCES ET TECHNOLOGIE

LICENCE SCIENCES & TECHNOLOGIES, 1re ANNÉE

UE INF201/231

ALGORITHMIQUE ET PROGRAMMATION FONCTIONNELLE

2022-2023

[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

EXERCICES DE TRAVAUX DIRIGÉS

Table des matières

Première partie

TYPES, EXPRESSIONS ET FONCTIONS

Chapitre 1

Appropriation des notations

Objectif Se familiariser avec

- la notation fonctionnelle,
- la vérification du typage d'une expression.

Les exercices qui suivent sont présentés sous forme de tableaux à compléter, constitués des colonnes : contexte, expression, type et valeur. Une ligne d'un tableau doit être comprise de la façon suivante : dans ce contexte, l'expression a tel type et telle valeur.

L'expression, la valeur et le type doivent être indiqués tantôt sous forme littérale, tantôt en Ocaml, tantôt sous les deux formes.

Rappel L'évaluation d'une expression dépend des valeurs associées aux noms qui apparaissent dans l'expression. Le *contexte* d'une expression désigne les associations nom-valeur qui donnent un sens à cette expression. Un contexte est un ensemble d'associations $nom \mapsto valeur$. En mathématiques (resp. en OCaml), on le définit grâce à la construction « *soit* $nom = valeur$ » (resp. `let nom = valeur`). Lorsqu'une expression peut être évaluée sans contexte, on dit que son contexte d'évaluation est vide dénoté par le symbole $\emptyset$ (ensemble vide).

Sommaire

Dans le titre de chaque exercice, problème ou annale d'examen est indiqué TDn ou TPn , où n est un entier entre 1 et 12 correspondant au numéro de séance de TD ou TP **à partir duquel** l'énoncé peut être abordé. En pratique, un exercice n^o n sera peut-être traité en séance $n+1$ ou $n+2$, et les exercices ne sont pas forcément traités dans l'ordre du polycopié.

Il y a plus d'énoncés que de séances ; ne pas hésiter à s'entraîner sur les énoncés qui n'auront pas été abordés en séance, notamment pendant le 2^e créneau de TP (non supervisé).

[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

1.1 TD1 Notation des types et des valeurs

Q1. Compléter le tableau ci-dessous :

Si besoin, on pourra utiliser une fonction de conversion :

SPÉCIFICATION 1	
PROFIL	`float_of_int` : $\mathbb{Z} \to \mathbb{R}$
SÉMANTIQUE	`(float_of_int x)` est le réel correspondant à l'entier `x`.

Cette fonction est prédéfinie dans la librairie standard d'OCaml.

1.2 TD1 Opérations logiques

En informatique, comme en mathématiques, la notion d'égalité apparaît dans différentes situations, et notamment :

- pour définir des liaisons nom $\mapsto$ valeur,
- pour effectuer des comparaisons entre valeurs.

En OCaml, ces deux situations sont dénotées par le même symbole : =. Dans les tableaux suivants, on prendra donc soin de ne pas confondre le = des colonnes «CONTEXTE» du = des colonnes «EXPRESSION».

Table de vérité des opérateurs logiques

Q1. Compléter le tableau ci-dessous :

	CONTEXTE	EXPRESSION	SÉMANTIQUE
N°		littérale $\leftrightarrow$ OCaml	OCaml
1	`let` $a = true$	a $\leftrightarrow$.	....
2	`let` $a = false$	a $\leftrightarrow$.	....
3	`let` $a = false$	$a =$ vrai $\leftrightarrow$	....
4	`let` $a = true$	$a =$ vrai $\leftrightarrow$	....
5	`let` $a = false$	$\neg a$ $\leftrightarrow$	....
6	`let` $a = true$	$\neg a$ $\leftrightarrow$	....
7	`let` $a = false$	$a =$ faux $\leftrightarrow$	....
8	`let` $a = true$	$a =$ faux $\leftrightarrow$	....

Considérons la table de vérité de la conjonction, notée $\wedge$ (`&&` en OCaml), usuellement appelée «et».

Q2. Compléter la table ci-dessous :

N°	CONTEXTE	EXPRESSION littérale	EXPRESSION ↔ OCAML	TYPE math	TYPE ↔ OCAML	SÉMANTIQUE littérale	SÉMANTIQUE ↔ OCAML
1	∅	3	↔ `3`	$\mathbb{Z}$	↔ `int`	3	↔ 3
2	∅	$-2{,}5+1{,}0$	↔ `-. 2.5.....`	$\mathbb{R}$	↔	.....	↔ .....
3	∅	$3{,}4+2$	↔	..	↔	.....	↔ .....
4	∅	¬ vrai	↔	.	↔	....	↔ .....
5	∅	$3 \neq 2$	↔ `3<>2`	.	↔	....	↔ .....
6	∅	2 + vrai	↔	.....		.....	
7	∅	'3'	↔	.....	↔	..	↔ ....
8	∅	'a'	↔	.....	↔ ..... (*minuscule*)	..	↔ ....
9	∅	','	↔ ...	.....	↔	...	↔ .....
10	∅	$'a' \leq 'b' \leq 'A'$	↔	.	↔	....	↔ .....
11	∅	x ou faux	↔	.	↔	.	↔ .
12	`let pi = 3.14`	$pi+1{,}0$	↔	..	↔	....	↔ .....
13	`let k = 3`	$2k$	↔ `2*k`	$2\mathbb{Z}$	↔ ....(*pair*)	.	↔ .

	CONTEXTE	EXPRESSION	SÉMANTIQUE
N°		litterale ↔ OCAML	OCAML
9	`let a = false and b = false`	a ∧ b ↔	.....
10	`let a = false and b = true`	a ∧ b ↔	.....
11	`let a = true and b = false`	a ∧ b ↔	.....
12	`let a = true and b = true`	a ∧ b ↔	.....

Considérons la table de vérité de la disjonction, notée ∨ (`||` en OCaml), usuellement appelée «ou».

Q3. Compléter la table ci-dessous :

	CONTEXTE	EXPRESSION	SÉMANTIQUE
N°	OCAML	litterale ↔ OCAML	OCAML
13	.....	a ∨ b ↔	....
14	.....	a ∨ b ↔	....
15	.....	a ∨ b ↔	....
16	.....	a ∨ b ↔	....

Identités remarquables

On suppose que le contexte est quelconque.

Q4. En observant les tables de vérité de $\neg a$ ($\neg$ est la négation, usuellement appelée «non»), $a =$ vrai et $a =$ faux, déduire une simplification de chacune des expressions suivantes :

expression litterale.	forme simplifiée
$a =$ vrai	.
$a =$ faux	...

Q5. Donner une expression équivalente (colonne de droite) de chacune des expressions suivantes (colonne de gauche) en utilisant les opérateurs $\neg$ et $\vee$, mais sans utiliser l'opérateur $\wedge$:

expression litterale ↔ OCAML	expression équivalente litterale ↔ OCAML
$\neg a \wedge \neg b$ ↔	..... ↔ .....
..... ↔	$\neg(\neg a \vee \neg b)$ ↔ .....

1.3 TD1 Opérations arithmétiques et logiques

Q1. Si l'expression est typable dans le contexte proposé, préciser son type et donner sa valeur.

N°	CONTEXTE	EXPRESSION	TYPE	VALEUR
1	`let a = 4 and b = 7`	`a + b`	`int`	`11`
2	`let a = 4 and b = 2.3`	`a + b`	.....	.....
3	`let a = 4 and b = true`	`a + b`	.....	.....
4	`let a = 7 and b = 4`	`a < b`	....	.....
5	`let a = 4 and b = 7 and c = 5`	`a - b - c`	...	...
6	`let a = 4 and b = 7 and c = 5`	`a < b < c`	.....	.....
7	`let a = 4 and b = 7 and c = 5 and d = 3`	`a < b && c < d`	....	.....
8	`let a = 4 and b = 7 and c = 5`	`(a > b && a < c) || a < b`	....	....
9	`let a = 4 and b = 7 and c = 5`	`a > b && (a < c || a < b)`	....	.....
10	`let a = 4 and b = 7 and c = 5`	`(a > b && a < c) || a > b`	....	.....
11	`let a = 4 and b = 7 and c = 5`	`(a > b || a <= b) && a < c`	....	....
12	.....	`(a < b) || (a > b)`	....	`true`
13	.....	`(a < b) || (a >= b)`	....	....

1.4 TD1 Chaînes de caractères

Dans le tableau suivant, le contexte est noté mathématiquement comme un ensemble de liaisons nom ↦ valeur plutôt qu'avec la syntaxe OCaml `let ... = ...`

Q1. Si l'expression est typable dans le contexte proposé, préciser son type et donner sa valeur.

N°	CONTEXTE (math.)	EXPRESSION (OCaml)	TYPE (math. ↔ OCaml)	VALEUR (OCaml)
1	$a \mapsto$ «*hibou*» $b \mapsto' x'$	`a ^ b`	..... ↔	.....
2	$a \mapsto$ «200» $b \mapsto 7$	`a ^ "b"`	..... ↔	.....
3	$a \mapsto$ «*hib*» $b \mapsto$ «*o*»	`a ^ b`	..... ↔	.....
4	$a \mapsto$ «␣» $b \mapsto$ «␣»	`a ^ b`	..... ↔	.....
5	$a \mapsto$ «*hib*» $b \mapsto$ «*o*»	`a @ b`	..... ↔	.....
6	$a \mapsto' a'$ $b \mapsto$ «*bc*»	`ab`	..... ↔	.....
7	$a \mapsto$ «*a*» $b \mapsto$ «*bc*»	`ab`	..... ↔	.....
8	$a \mapsto' a'$ $b \mapsto$ «*b*»	`"ab"`	..... ↔	.....
9	$a \mapsto$ «*B*» $b \mapsto$ «*A*»	`a ^ b ^ b ^ a`	..... ↔	.....
10	$a \mapsto$ "␣" $b \mapsto$ «*AB*»	`a ^ b ^ b ^ a`	..... ↔	.....
11	$a \mapsto$ «*a*» $b \mapsto$ « ^ »	`a b a`	..... ↔	.....

1.5 TD1 Expressions conditionnelles

L'expression `if … then … else …`

`if … then … else …` est une expression qui a un type et une valeur et donc peut être utilisée au cœur d'une autre expression !

Q1. Compléter le tableau suivant :

N°	CONTEXTE	EXPRESSION	TYPE	VALEUR
1	`let a = 4 and b = 7`	`if a < b then 3 else 4`	`int`	.
2	`let a = 4 and b = 7`	`(if a < b then 3 else 4) + a`	...	.
3	`let a = 4 and b = 7`	`if a < b then 3 else (4 + a)`	...	.

Un `if` sans `else` ?

On considère la fonction suivante :

SPÉCIFICATION 1 — valeur absolue

PROFIL $abs : \mathbb{Z} \to \mathbb{N}$

SÉMANTIQUE $abs\ (e)$ est la valeur absolue de e.

EX. ET PROP. (i) $abs\ (-3) = 3$

RÉALISATION 1

ALGORITHME expression conditionnelle déterminant le signe de e

IMPLÉMENT.
```
let abs (e:int) : int =
  if e<0 then -e
```

Q2. Pourquoi cette implémentation est-elle incorrecte ?

Q3. Donner une implémentation correcte.

Expressions conditionnelles à valeur booléenne

On suppose que le contexte est quelconque.

Q4. Donner une expression OCaml équivalente de chacune des expressions suivantes, en utilisant uniquement les opérateurs logiques :

N°	EXPRESSION	EXPRESSION ÉQUIVALENTE OCAML
1	`if a then true else false`	.
2	`if a then false else true`	.....
3	`if a then b else false`	.....
4	`if a then true else b`	.....
5	`if not a then true else b`	.....

1.6 TD1 Vérification des types dans une expression

On s'intéresse à la vérification des types des noms qui apparaissent dans une expression algébrique par rapport aux spécifications des opérations qu'elles mettent en jeu.

Par exemple, pour que l'expression `if a then 1. else x` soit correcte du point de vue des types, les contraintes suivantes doivent être respectées :

- `a` doit être de type booléen,
- `x` doit être de type réel.

Ces contraintes sont la conséquence du fait que l'expression qui suit le `if` doit être de type booléen et que la construction `if then else` étant une expression, ses deux branches doivent avoir le même type. Si ces conditions sont respectées, l'expression `if a then 1. else x` est de type réel.

Autre exemple : dans l'expression `a = b`, la comparaison n'a de sens que si `a` et `b` sont de même type, disons t. Si cette contrainte est respectée, l'expression est de type booléen.

Types de base

Q1. Pour chacune des expressions ci-dessous, donner les contraintes de types que doivent respecter les noms y apparaissant, puis le type de l'expression si ces contraintes sont respectées, sinon proposer une correction de l'expression.

N°	EXPRESSION (OCAML)	CONTRAINTE	TYPE
1	$a \vee b$	....	..
2	`if a then (x && y) else t`	....	....
3	`3 + (if x=y+1 then a else b)`	....	....
4	$(a < b) \wedge (c < d)$	....	..
5	$a = (b <= c)$	....	..
6	`if not b then 5. else z`	....	..

Types construits, fonctions

Q2. Pour chacune des expressions ci-dessous, donner les contraintes de types que doivent respecter les noms y apparaissant, puis le type de l'expression si ces contraintes sont respectées.

- `(a+.1., not b, 5)`

- `let c=2 and d=4. in (c+3, d)`
- `if a && g then (a,b) else (x,g)`
- `let f x = x+.1. in f 5.`
- ```
  let h (u) (v) (z) = match u with
      | true -> if a=b then 7. else v
      | false -> z
  ```
  ```

Chapitre 2

Types, expressions et fonctions

Sommaire

Dans le titre de chaque exercice, problème ou annale d'examen est indiqué TDn ou TPn , où n est un entier entre 1 et 12 correspondant au numéro de séance de TD ou TP **à partir duquel** l'énoncé peut être abordé. En pratique, un exercice n^o n sera peut-être traité en séance $n+1$ ou $n+2$, et les exercices ne sont pas forcément traités dans l'ordre du polycopié.

Il y a plus d'énoncés que de séances ; ne pas hésiter à s'entraîner sur les énoncés qui n'auront pas été abordés en séance, notamment pendant le 2^e créneau de TP (non supervisé).

2.1 TD1 Signe du produit

Étant donnés deux entiers, on veut – **sans** calculer leur produit – déterminer si le produit des deux nombres est positif ou nul ou bien s'il est strictement négatif.

Q1. Compléter ci-dessous la spécification de la fonction *prodPos* qui répond à cette demande :

SPÉCIFICATION 1 — signe du produit	
PROFIL	*prodPos* : .....
SÉMANTIQUE	(*prodPos x y*) est vrai si et seulement si le signe du produit xy est positif ou nul

EX. ET PROP. (i) $(prodPos\ -2\ -3) = \ldots$.

(ii) $\forall x \in \mathbb{Z}, (prodPos\ x\ 0) = \ldots$.

(iii) ..... = faux

Q2. Compléter chacune des implémentations suivantes de la fonction *prodPos*. On prendra soin :

- d'indenter le code,
- de préciser en commentaire quelles sont les expressions booléennes au niveau des `else`.

RÉALISATION 1 — signe du produit

ALGORITHME a) analyse par cas dirigée par le résultat de la fonction et composition conditionnelle

IMPLÉMENT.
```
let prodPos_1 ..... =                                    1
    if ..... then                                        2
        true                                             3
    else ...........................................     4
       false                                             5
```

ALGORITHME b) analyse par cas dirigée par les données de la fonction et composition conditionnelle

IMPLÉMENT.
```
let prodPos_2 ..... =                                    1
    if x > 0 then                                        2
    ...................................                  3
    ...................................                  4
    ...................................                  5
    ...................................                  6
    else (* x ≤ 0 *)                                     7
    ...................................                  8
    ...................................                  9
    ...................................                  10
    ...................................                  11
    ...................................                  12
    ...................................                  13
    ...................................                  14
```

Q3. L'étudiant JeSuisPlusMalin propose une implémentation *prodPos_2'* identique à *prodPos_2* mais sans les lignes 9, 13 et 14. Son implémentation respecte-elle la spécification de *prodPos* ? Si non, donner un contre-exemple.

2.2 TD1 Une date est-elle correcte ?

On considère une date représentée par deux entiers j et m : j est le numéro du jour dans le mois et m est le numéro du mois dans l'année. On veut déterminer si les deux entiers correspondent à

une date valide d'une année non bissextile[1]. On spécifie pour cela les types *jour* et *mois* ainsi qu'une fonction *estJourDansMois* :

DÉFINITION D'ENSEMBLES

déf $jour = \{1, \ldots, 31\}$

déf $mois = \{1, \ldots, 12\}$

SPÉCIFICATION 1 — jour d'un mois	
PROFIL	*estJourDansMois* : .....
SÉMANTIQUE	(*estJourDansMois j m*) est *vrai* si et seulement si *j* et *m* caractérisent respectivement le jour et le mois d'une date d'une année non bissextile.
EX. ET PROP.	(i) (*estJourDansMois* 28 1) = ...
	(ii) (*estJourDansMois* 31 4) = ...
	(iii) (*estJourDansMois* 18 13) n'a pas de sens puisque .....
	(iv) (*estJourDansMois* 0 4) n'a pas de sens puisque .....

Q1. Compléter la spécification de la fonction *estJourDansMois* ci-dessus, et l'implémentation des ensembles *jour* et *mois* ci-dessous :

```
type jour = .... (* restreint à ..... *)   1
type mois = .... (* restreint à ..... *)   2
```

Q2. Compléter la réalisation de *estJourDansMois* avec deux implémentations différentes basées sur les algorithmes suivants :

RÉALISATION 1 — jour d'un mois	
ALGORITHME	1) composition conditionnelle sous forme d'expressions conditionnelles imbriquées examinant successivement les 3 cas : mois à 31, mois à 28, mois à 30.
IMPLÉMENT.	. . .
ALGORITHME	2) composition booléenne examinant successivement les 3 cas : mois à 31, mois à 28, mois à 30. L'utilisation d'expressions conditionnelles est interdite.
IMPLÉMENT.	. . .

2.3 TD1 Quelle heure est-il ?

Le type *horaire*. On étudie la représentation de l'heure affichée par une montre en mode AM/PM. Les heures sont limitées à l'intervalle $\{0, \ldots, 11\}$.

[1] Le mois de février des années non bissextiles a 28 jours.

L'indication *am*/*pm* indique si l'heure du jour est située entre minuit et midi (avant midi, en latin *ante meridiem*, abrégé en *am*) ou entre midi et minuit (après midi, en latin *post meridiem*, abrégé en *pm*).

On choisit de représenter une date horaire par un quadruplet formé de trois entiers et la valeur Am ou Pm de type énuméré *meridien*.

Les entiers représentent les heures, minutes et secondes.

On définit ainsi les ensembles suivants :

- $heure \stackrel{def}{=} \{0, \dots, 11\}$
- $minute \stackrel{def}{=} \{0, \dots, 59\}$
- $seconde \stackrel{def}{=} \{0, \dots, 59\}$
- $meridien \stackrel{def}{=} \{\text{Am}, \text{Pm}\}$
- $horaire \stackrel{def}{=} heure \times minute \times seconde \times meridien$

Exemples :

- Le quadruplet $(3, 0, 0, \text{Pm})$ représente l'heure exprimée habituellement par 3 : 00 : 00 PM.
- Le quadruplet $(2, 25, 30, \text{Am})$ représente l'heure 2 : 25 : 30 AM.
- Noter que $(0, 0, 0, \text{Am})$ correspond à minuit et que $(0, 0, 0, \text{Pm})$ correspond à midi.

Incrémenter l'horaire d'une seconde. On considère la fonction suivante :

SPÉCIFICATION 1 — incrémenter d'une seconde	
PROFIL	*inc_hor* : *horaire* → *horaire*
SÉMANTIQUE	*inc_hor* (*h*) est l'horaire qui suit d'une seconde l'horaire *h*.
EX. ET PROP.	(i) *inc_hor* $(2, 25, 30, \text{Am}) = (2, 25, 31, \text{Am})$ (ii) *inc_hor* $(11, 59, 59, \text{Pm}) = (0, 0, 0, \text{Am})$

Q1. Implémenter les types *heure*, *minute*, *seconde*, *meridien* et *horaire*, puis donner une réalisation de la fonction *inc_hor*.

Q2. Utiliser la fonction *inc_hor* pour réaliser la fonction *ajout3sec* :

SPÉCIFICATION 2	
PROFIL	*ajout3sec* : *horaire* → *horaire*
SÉMANTIQUE	*ajout3sec* (*h*) est l'horaire qui vient 3 secondes après *h*.

2.4 TD2 Fractions

On étudie un type correspondant à la notion de fraction positive ou nulle. On veut pouvoir :

- construire des fractions irréductibles à partir de deux entiers correspondant au numérateur et au dénominateur,

- disposer de certaines opérations sur les fractions.

Il s'agit de spécifier le type *fraction* et une fonction de construction associée. Un objet de type *fraction* est un couple d'entiers formé du numérateur et du dénominateur et tels que leur plus grand diviseur commun (*pgcd*) vaut 1 ; la fraction est donc sous forme irréductible :

$$\textbf{\textit{déf}} \quad fraction = \{(n,d) \in \mathbb{N} \times \mathbb{N}^* \mid pgcd(n,d) = 1\}$$

Q1. Implémenter le type *fraction*.

La fonction de construction d'un objet de type *fraction*, nommée *frac* a pour paramètres deux entiers *num* et *den* correspondants à la réduction de la fraction $\frac{num}{den}$.

EX. ET PROP.
(i) $(frac\ 21\ 60)$ construit le couple correspondant à la réduction de la fraction $\frac{21}{60}$: $(frac\ 21\ 60) = (7,20)$
(ii) $(frac\ 42\ 120) = (7,20)$
(iii) $(frac\ 7\ 20) = (7,20)$

Pour mettre une fraction sous forme irréductible, on dispose d'une fonction de calcul du plus grand diviseur commun de deux entiers, dont le profil est le suivant : $pgcd : \mathbb{Z}^* \to \mathbb{Z}^* \to \mathbb{Z}^*$

Q2. Compléter la définition de la fonction *frac* ci-dessous :

SPÉCIFICATION 1 — construction de fractions	
PROFIL	*frac* : .....
SÉMANTIQUE	.....
RÉALISATION 1	
ALGORITHME	.....
IMPLÉMENT.	`/!\ conditions utilisation : .....`

2.5 TD2 Géométrie élémentaire

Un point dans le plan Euclidien est repéré par ses coordonnées (abscisse et ordonnée).

Q1. Définir les ensembles mathématiques associés aux abscisses, ordonnées et coordonnées d'un point.

Q2. Donner la spécification de la fonction *longueur* qui calcule distance entre deux points.

Q3. Donner une réalisation de *longueur*, la fonction « racine carrée » – prédéfinie en OCaml – ayant le profil suivant : $sqrt : \mathbb{R}^+ \to \mathbb{R}^+$. La réalisation de *sqrt* n'est pas demandée.

On souhaite manipuler des figures géométriques telles que :

- les *triangles*, définis par leurs trois sommets ;
- les *cercles*, définis par leur centre et leur rayon ;

- les *rectangles*, dont les cotés sont parallèles aux axes, définis par leur coin inférieur gauche, et leur coin supérieur droit.

On définit pour cela l'ensemble $figure$ suivant :

$$\begin{aligned} \textbf{\textit{déf}}\ \ figure \ \ = \ \ & \{\ \textsc{Triangle}(t) \mid t \in \ldots\ldots\ldots\ldots\ldots\ldots\ \} \cup \\ & \{\ \textsc{Cercle}(.) \mid \ldots\ldots\ldots\ldots\ldots\ \} \cup \\ & \{\ \ldots\ldots\ldots\ldots\ \mid \ldots\ldots\ldots\ldots\ldots\ \} \end{aligned}$$

Q4.

a) Que représente le terme TRIANGLE dans cette définition ?

b) Qu'est-ce-que t ? Quel est son type ? Comment appelle-t-on un tel type ?

c) Compléter la définition du type $figure$ ci-dessus.

d) Donner une implémentation de ce type.

Q5. Donner la spécification de la fonction *peri* qui calcule le périmètre d'une figure.

Q6. Réaliser la fonction *peri* sans oublier d'indiquer l'algorithme retenu.

2.6 TD2 Relations sur des intervalles d'entiers

On caractérise un intervalle **fermé** d'entiers par ses bornes inférieure et supérieure :

$$intervalle \stackrel{def}{=} \mathbb{Z} \times \mathbb{Z}$$

Soit $bi, bs \in \mathbb{Z}$; le couple (bi, bs) décrit un intervalle d'entiers de borne inférieure bi et de borne supérieure bs, à condition que la contrainte $bi \leq bs$ soit respectée.

2.6.1 Relations entre points et intervalles

Pour situer un entier par rapport à un intervalle, on spécifie les trois fonctions suivantes :

SPÉCIFICATION 1 — prédicats sur les intervalles	
PROFIL	$dans\ :\ \mathbb{Z} \to intervalle \to \mathbb{B}$
SÉMANTIQUE	$(dans\ x\ i) = vrai \ \Leftrightarrow\ x \in i$
PROFIL	$precede\ :\ \mathbb{Z} \to intervalle \to \mathbb{B}$
SÉMANTIQUE	$(precede\ x\ i)$ si et seulement si x est strictement inférieur aux entiers de i
PROFIL	$suit\ :\ \mathbb{Z} \to intervalle \to \mathbb{B}$
SÉMANTIQUE	$(suit\ x\ i)$ si et seulement si x est strictement supérieur aux entiers de i

Q1. Quels seraient les profils de ces trois fonctions si le type *intervalle* n'avait pas été défini ? Remarquer comment cette définition facilite la compréhension des profils des fonctions.

Q2. Compléter la spécification donnée ci-dessus par des exemples.

Q3. Réaliser les fonctions *precede*, *dans* et *suit* sans composition conditionnelle (l'utilisation de `if_then_else` est interdite).

2.6.2 Relations entre intervalles

Q4. Compléter la spécification de la fonction suivante :

SPÉCIFICATION 2 — relations sur les intervalles	
PROFIL	*coteAcote* : *intervalle* → *intervalle* → $\mathbb{B}$
SÉMANTIQUE	*coteAcote* (i_1, i_2) est vrai si et seulement si i_1 et i_2 sont contigus sans se toucher.
EX. ET PROP.	(i) (*coteAcote* $(1,3)$ $(4,10)$) = *vrai* (ii) (*coteAcote* $(2,8)$ $(-1,1)$) = *vrai* (iii) (*coteAcote* $(1,3)$ $(3,4)$) = ... (iv) (*coteAcote* $(2,8)$ $(3,4)$) = ... (v) cas où les intervalles ont une intersection vide : (*coteAcote* .....

Q5. Implémenter la fonction *coteAcote* sans composition conditionnelle (l'utilisation de `if_then_else` est interdite) ; donner des exemples d'utilisation pertinents.

SPÉCIFICATION 3 — relations sur les intervalles (suite)	
PROFIL	*chevauche* : *intervalle* → *intervalle* → $\mathbb{B}$
SÉMANTIQUE	(*chevauche* i_1 i_2) est vrai si et seulement si i_1 et i_2 n'ont pas de bornes communes, aucun n'est inclus dans l'autre mais ils ont une intersection non vide.

Q6. Implémenter la fonction *chevauche* sans composition conditionnelle (utilisation de `if_then_else` interdite) ; donner des exemples d'utilisation pertinents.

2.7 TD3 Le code de César

On désire coder un texte ne comportant que des lettres majuscules et des espaces, en remplaçant chaque caractère par un autre caractère selon les règles suivantes (code dit « de César ») :

- à un espace correspond un espace,
- à la lettre `'A'` correspond la lettre `'D'`, à `'B'` correspond `'E'`, ..., à `'W'` correspond `'Z'`, à `'X'` correspond `'A'`, à `'Y'` correspond `'B'`, à `'Z'` correspond `'C'`.

Inversement, on veut pouvoir décoder un texte codé selon ce qui précède. Pour cela on étudie une fonction de codage, *codeCar*, et sa fonction réciproque, *decodeCar*.

Q1. Spécifier (profil, sémantique, exemples et propriétés) les fonctions *codeCar* et *decodeCar*.

2.7.1 Version 1

Q2. Expliquer brièvement comment procéder à la main pour coder ou décoder un caractère.

Q3. En se limitant à un alphabet de 5 lettres $\{A, B, C, D, E\}$, implémenter chacune des fonctions *codeCar* et *dcodeCar* par composition conditionnelle sous forme d'expressions conditionnelles imbriquées.

Q4. Proposer une autre implémentation en utilisant le filtrage

2.7.2 Version 2

Le code de César est fondé sur un décalage sur un alphabet considéré comme circulaire, c'est-à-dire que les lettres sont rangées dans l'ordre alphabétique et que la dernière lettre de l'alphabet est suivie par la première lettre :

`'A'` puis `'B'` puis `'C'` puis ...puis `'Z'` puis `'A'` ...

Le code d'une lettre est la lettre située 3 positions après elle dans l'ordre défini ci-dessus.

Idée *Assurer le codage en associant à chaque lettre un entier, par exemple son rang dans l'alphabet et utiliser des opérations sur les entiers pour exprimer le codage ou le décodage.*

Cette décomposition du problème est illustrée par le schéma suivant :

$$l \xrightarrow{codeCar} l'$$
$$l \xrightarrow{rangCar} r \xrightarrow{plus3} r' \xrightarrow{iemeCar} l'$$

- l dénote une lettre quelconque et l' dénote le code de l ; r désigne l'entier associé à l et r' celui associé à l'.
- Les flèches sont étiquetées par un nom de fonction. Elles montrent la manière de réaliser la fonction *codeCar* par composition fonctionnelle.

Les trois fonctions intermédiaires qui apparaissent sur le schéma sont spécifiées comme suit :

$$nat1\_26 \stackrel{def}{=} \{1, \dots, 26\}$$

SPÉCIFICATION 2	
PROFIL	*plus*3 : *nat1_26* → *nat1_26*
SÉMANTIQUE	*plus*3 (r) décale le rang r de 3 de manière circulaire dans $\{1, \dots, 26\}$
PROFIL	*rangCar* : *majuscule* → *nat1_26*
SÉMANTIQUE	*rangCar* (l) est le rang dans l'alphabet de la lettre l donnée
PROFIL	*iemeCar* : *nat1_26* → *majuscule*
SÉMANTIQUE	*iemeCar* (r) est la lettre de rang r dans l'alphabet

Q5. Compléter la spécification par des exemples.

Q6. Implémenter le type *nat1_26* et la fonction *plus*3.

Q7. En utilisant le code ASCII de 'A' définit comme suit, implémenter *iemeCar* et *rangCar* :
`let cst_ASCII_A = int_of_char('A')`
où `int_of_char` est la fonction prédéfinie dans la librairie standard d'OCaml :

SPÉCIFICATION	
PROFIL	`int_of_char` : *char* → {0, … , 255}
SÉMANTIQUE	`(int_of_char c)` est le code ASCII de *c*.

Q8. Implémenter *codeCar*

Q9. Donner une réalisation de la fonction *decodeCar* en se basant sur un principe analogue à ce qui a été fait pour *codeCar*.

2.7.3 Généralisation

On généralise le principe du codage en se basant sur un décalage de d positions dans l'alphabet ($d = 3$ étant le cas particulier précédent). On notera qu'il est inutile d'envisager des décalages de plus de 25 ; on pourra à cette fin utiliser l'ensemble $\{1, \dots, 25\}$.

Q10. Spécifier les fonctions généralisées de codage *codeCarGen* et de décodage *decodeCarGen*, puis donner leur réalisation en utilisant les fonctions intermédiaires adéquates.

Q11. Réimplémenter *codeCar* en utilisant uniquement *codeCarGen*.

2.8 TD3 À propos de dates

Parmi les diverses formes d'expression de la date d'un jour dans le calendrier grégorien, nous nous intéressons à la suivante : une date est caractérisée par un triplet d'entiers composé du quantième du jour dans le mois, du quantième du mois dans l'année et de l'année. Ainsi, le triplet $(22, 9, 1990)$ caractérise le 22ème jour du 9ème mois de l'année 1990.

On ne considère ici que des dates prises dans la période $\{1900, \dots, 2100\}$. On rappelle qu'une année est dite *bissextile* si son expression numérale est divisible par 4, comme 1968, 1972, 1976. Toutefois les années séculaires[2] ne sont pas bissextiles, sauf celles dont les deux premiers chiffres forment un nombre divisible par 4, comme 1600, 2000, 2400.

Dans ce qui suit, on étudie diverses fonctions sur les dates.

2.8.1 Caractéristiques d'une date grégorienne

La période considérée est caractérisée par ses années de début et de fin. Une année est dénotée par un entier compris entre ces deux années.

- Le mois est dénoté par un entier compris entre 1 et 12.
- Un jour dans une année est désigné par un entier compris entre 1 et 365 ou 366 selon que l'année est bissextile ou non.

[2] *séculaire* signifie « de changement de siècle »

- Un jour dans le mois est représenté par un entier dans un intervalle qui dépend du mois considéré, mais qui est toujours inclu dans l'intervalle $\{1, \ldots, 31\}$.

On définit les constantes et les types suivants :

- $\text{AnDéb} \stackrel{def}{=} 1900$, $\text{AnFin} \stackrel{def}{=} 2100$
- $quantieme \stackrel{def}{=} \{1, \ldots, 366\}$
- $jour \stackrel{def}{=} \{1, \ldots, 31\}$, $mois \stackrel{def}{=} \{1, \ldots, 12\}$
- $anne \stackrel{def}{=} \{\text{AnDéb}, \ldots, \text{AnFin}\}$
- $date \stackrel{def}{=} jour \times mois \times anne$

On dispose d'une fonction *ent_en_ch* spécificée ainsi :

SPÉCIFICATION 1	
PROFIL	`string_of_int` : $\mathbb{Z} \to string$
SÉMANTIQUE	`string_of_int`(n) est la chaine correspondant à l'entier n
EX. ET PROP.	(i) (*string_of_int* 125) = `"125"` (ii) (*string_of_int* 5) = `"5"` (iii) (*string_of_int* 0) = `"0"`

Cette fonction est prédéfinie dans la librairie standard d'OCaml.

2.8.2 Quantième d'une date dans son année

On étudie une fonction *date_en_quant* :

SPÉCIFICATION 2	
PROFIL	$date\_en\_quant : jour \times mois \times anne \to quantieme$
SÉMANTIQUE	$date\_en\_quant(j, m, a)$ est le numéro du jour dans l'année
EX. ET PROP.	(i) $date\_en\_quant(15, 11, 1993) = 319$

Pour déterminer le quantième d'une date dans son année, on commence par calculer le quantième du premier jour du mois donné dans le cas d'une année non bissextile. On définit pour cela la fonction :

SPÉCIFICATION 3	
PROFIL	$mois\_en\_quant : mois \to quantieme$
SÉMANTIQUE	$mois\_en\_quant(m)$ est le quantième du *premier jour* du mois m dans une année non bissextile.
EX. ET PROP.	(i) $mois\_en\_quant(11) = 304$

Q1.

a) Réaliser la fonction *date_en_quant* dans le cas d'une année non bissextile à l'aide de la fonction *mois_en_quant*.

b) Compléter la réalisation précédente pour tenir compte des années bissextiles. On définit pour cela le prédicat :

SPÉCIFICATION 4	
PROFIL	$est\_biss : anne \to \mathbb{B}$
SÉMANTIQUE	(*est_biss* a) est vrai si et seulement si a est une année bissextile.

c) Réaliser la fonction *est_bis* en tenant compte de la période considérée {AnDéb, ... , AnFin}.

2.8.3 Affichage d'une date

On considère la fonction :

SPÉCIFICATION 5	
PROFIL	$date\_en\_ch : date \to string$
SÉMANTIQUE	$date\_en\_ch(d)$ est la chaine représentant la date d
EX. ET PROP.	(i) $date\_en\_ch(15, 11, 1993) =$ « 15 novembre 1993 » (ii) $date\_en\_ch(1, 11, 1993) =$ « 1er novembre 1993 »

Pour réaliser *date_en_ch* on introduit une fonction :

SPÉCIFICATION 6	
PROFIL	$mois\_en\_ch : mois \to string$
SÉMANTIQUE	$mois\_en\_ch(m)$ est le nom du mois m sous forme de chaine.

Q2. Réaliser la fonction *date_en_ch* en utilisant la fonction *mois_en_ch*.

2.8.4 Date du lendemain

Soit la fonction :

SPÉCIFICATION 7	
PROFIL	$lendemain\_en\_ch : date \to string$
SÉMANTIQUE	$lendemain\_en\_ch(d)$ est la chaine représentant le jour qui suit la date d ou bien « inconnue » si d est la dernière date de la période

Pour réaliser *lendemain_en_ch* on introduit une fonction :

SPÉCIFICATION 8	
PROFIL	$inc\_date : date \to date$
SÉMANTIQUE	$inc\_date(d)$ est la date du jour qui suit d

Pour déterminer la date du lendemain, il faut savoir si c'est la fin d'un mois et pour cela connaître le dernier jour de chaque mois :

SPÉCIFICATION 9	
PROFIL	$est\_fin\_mois : date \to \mathbb{B}$
SÉMANTIQUE	$est\_fin\_mois(d)$ vaut *vrai* si et seulement si la date (j, m, a) est le dernier jour du mois m

SPÉCIFICATION 10	
PROFIL	$dernier\_jour : mois \to jour$
SÉMANTIQUE	$dernier\_jour(m)$ est le numéro du dernier jour du mois m dans le cas d'une année non bissextile

Q3.

a) Réaliser dans cet ordre les fonctions *lendemain_en_ch*, *inc_date*, *est_fin_mois*.

b) Réaliser dans cette ordre les fonctions *mois_en_ch*, *dernier_jour* et *mois_en_quant*.

2.9 TD3 Nomenclature de pièces

On considère ici un problème d'informatisation d'un stock de pièces d'un magasin de fournitures pour automobiles, et plus particulièrement la manière de référencer ces pièces. Une nomenclature rassemble l'ensemble des conventions permettant de regrouper les pièces en différentes catégories et d'associer à chaque pièce une référence unique servant à la désigner. Ces conventions sont ici les suivantes :

- À un 1er niveau, les pièces sont réparties selon la matière dans laquelle elles sont fabriquées :

$$matiere \stackrel{def}{=} \{\text{Zinc}, \text{Plomb}, \text{Fer}, \text{Cuivre}, \text{Alu}, \text{Carbone}, \text{Autres}\}$$

- À un 2e niveau, on différencie les pièces d'une même matière par un entier entre 0 et 999.

$$numero \stackrel{def}{=} \{0, \dots, 999\}$$

Une réference est définie par 5 caractères : les deux premiers correspondent à deux lettres de la matière de la pièce et les trois caractères suivants sont le numéro de la pièce (avec des zéros en tête si nécessaire). Exemples :

- le nom AL053 désigne la pièce de numéro 053 dans la famille des pièces en aluminium
- le nom CA053 désigne la pièce de numéro 053 dans la famille des pièces en carbone

2.9.1 Types associés à la notion de référence

Pour représenter les références et leurs noms, on définit les ensembles *reference* et *nom* :

$$reference \stackrel{def}{=} matiere \times numero$$
$$nom \stackrel{def}{=} majuscule \times majuscule \times chiffre \times chiffre \times chiffre$$

Exemple : le couple (Alu, 53) est la valeur de type *reference* correspondant au nom `('A','L','0','5','3')`. Inversement, le nom `('C','A','0','5','3')` est représenté informatiquement par la référence (Carbone, 53).

Q1. Implémenter les types *matiere*, *numero*, *reference* et *nom*.

2.9.2 Fonctions associées à la notion de référence

On considère une fonction, nommée *nom_en_ref*, spécifiée comme suit :

SPÉCIFICATION 1 — conversion d'un nom en référence	
PROFIL	$nom\_en\_ref : nom \to reference$
SÉMANTIQUE	$nom\_en\_ref\,(n)$ est la valeur de type *reference* associée au nom n.
EX. ET PROP.	(i) nom_en_ref `('C','A','0','5','3')` = (Carbone, 53) (ii) nom_en_ref `('A','L','0','5','3')` = (Alu, 53)

Pour réaliser la fonction *nom_en_ref*, on introduit les fonctions *chiffreVbase10* et *cc_en_mat* :
déf $base10 = \{0, ..., 9\}$

SPÉCIFICATION 2	
PROFIL	$chiffreVbase10 : chiffre \to base10$
SÉMANTIQUE	$chiffreVbase10\,(c)$ est l'entier associé au caractère décimal c.
EX. ET PROP.	(i) ($chiffreVbase10$ `'2'`) = 2 (ii) ($chiffreVbase10$ `'0'`) = 0
PROFIL	$cc\_en\_mat : caractère \times caractère \to matiere$
SÉMANTIQUE	$cc\_en\_mat\ (c_1, c_2)$ est la valeur de type *matiere* dont le nom commence par le caractère c_1 suivi de c_2 et la valeur Autres si aucune matière ne correspond.
EX. ET PROP.	(i) cc_en_mat `('C','A')` = Carbone (ii) cc_en_mat `('C','U')` = Cuivre (iii) cc_en_mat `('X','P')` = Autres (iv) cc_en_mat `('B','Z')` = Autres

Q2. Donner la réalisation de la fonction *nom_en_ref* en utilisant *cc_en_mat* et *chiffreVbase10*.

Q3. Donner deux implémentations de la fonction *chiffreVbase10* :

RÉALISATION 2	
ALGORITHME	1) composition conditionnelle, sous forme d'expressions conditionnelles imbriquées.

IMPLÉMENT.	.
	.
	.
ALGORITHME	2) composition conditionnelle, sous forme de filtrage
IMPLÉMENT.	.
	.
	.

NB : vous avez vu (ou vous verrez) en TP une implémentation de la fonction *chiffreVbase10* qui n'utilise pas d'expression conditionnelle.

Q4. Implémenter la fonction *cc_en_mat*.

2.9.3 Comparaison de références : relation d'ordre sur les types

Les opérateurs OCaml de comparaison (=, <>, <, <=, >, >=) sont polymorphes et s'appliquent à tous les types OCaml de base (dont les booléens, les caractères et les nombres), ainsi qu'aux types construits par composition.

Pour un type somme, l'ordre OCaml entre deux valeurs est défini implicitement par l'ordre entre les constructeurs du type (et si les valeurs partagent le même constructeur par l'ordre sur l'eventuel paramètre du constructeur). Dans la définition d'un type somme, les constructeurs sont énumérés par ordre croissant de valeur.

L'ordre implicite entre deux n-uplets d'un même type produit est l'ordre de leurs premiers éléments distincts de même rang (en partant de la gauche) : le premier élément d'un type produit est donc prépondérant dans une comparaison.

Q1. L'ordre implicite sur les noms correspond-t-il à l'ordre implicite sur les références ?

On définit *ordrebis*, un nouvel ordre sur les références, dans lequel la valeur relative des numéros prime sur la matière.

Q2. Implémenter la fonction suivante :

SPÉCIFICATION 3	
PROFIL	$inf\_ref\ :\ reference \to reference \to \mathbb{B}$
SÉMANTIQUE	$(inf\_ref\ r_1\ r_2)$ vaut *vrai* si et seulement si r_1 est avant r_2 suivant *ordrebis*.
EX. ET PROP.	(i) $(inf\_ref\ (Carbone, 13)\ (Zinc, 45)) = vrai$ (ii) $(inf\_ref\ (PLOMB, 12)\ (ALU, 943)) = vrai$

Chapitre 3

Annales

Il est fortement recommandé de réviser chaque partie en cherchant à résoudre un ou deux énoncés d'annales.

Sommaire

3.1 Partiel 2014–2015 TD3 «un dîner presque parfait»

Le but de l'exercice est de calculer le prix d'un repas, sachant que :

- un repas est constitué d'une entrée, plat principal, fromage et/ou dessert ;
- il y a trois tailles pour l'entrée ;
- le fromage se prend à l'unité ou au plateau ;
- certains plats nécessitent un supplément.

3.1.1 Modélisation

On definit les types OCaml suivants :

```
(* Trois plats principaux possibles : *)
type platPrincipal = P1 | P2 | P3

(* Une seule entrée possible, mais en trois tailles : *)
type tailleEntree = Petite | Moyenne | Grande

(* x morceaux de fromage à l'unité : M(x), ou au plateau : P *)
type fromage = M of int | P

(* Trois desserts possibles : *)
type dessert = D1 | D2 | D3
```

On supposera que les coûts des différents éléments du repas sont toujours des entiers naturels ($\mathbb{N}$).

Q1. (0,5pt) Implémenter en OCaml un type appelé `cout` représentant une somme entière d'argent. On prendra soin de préciser toute contrainte éventuelle sur les éléments de ce type.

3.1.2 Coûts des éléments du repas

Une petite entrée coûte 3 €, une moyenne 5 et une grande 7.

Q2. (1pt) Implémenter une fonction *coutEntree* qui renvoie le coût d'une entrée en fonction de sa taille.

Les plats principaux sont à 10 €, avec un *supplément* de 3 € pour `P3`. On définit donc les constantes suivantes :

```
let cstPRIXPRINCIPAL: cout = 10
let cstSUPLP3: cout = 3
```

Q3. (2pt) Spécifier (profil, sémantique, exemples ou propriétés) puis réaliser une fonction *coutPrincipal* qui renvoie le coût d'un plat principal.

Un morceau de fromage à l'unité coûte 2 € tandis que le plateau coûte 6 €.

Q4. (1,5pt)

a) Définir des constantes représentant ces différents coûts, et implémenter une fonction *coutFromage* qui renvoie le coût du fromage.

b) Donner une expression OCaml permettant de vérifier qu'il vaut mieux, d'un point de vue financier, prendre le plateau que quatre fromages.

Q5. (0,5pt) Proposer une implémentation en OCaml de la règle : « les desserts sont tous à 5 € » sous forme d'une fonction.

3.1.3 À table

On suppose qu'il existe deux formules pour les repas :

- $Formule_1$: une entrée et un plat,
- $Formule_2$: un plat et un dessert ou un fromage.

On donne la définition mathématique de l'ensemble « fromage ou dessert » :

$$fromOUdess \stackrel{def}{=} \{D(d) \ / \ d \in dessert\} \cup \{F(f) \ / \ f \in fromage\}$$

Q6. (2pt)

a) Quel est la nature de D et F ?

b) Implémenter $fromOUdess$ en OCaml

c) Quel est le profil (*ensemble de départ* → *ensemble d'arrivée*[1]) de `D` ? De `F` ?

Un repas est implémenté en OCaml de la manière suivante :

[1] ou, plus précisément, *domaine* → *codomaine*

```
type repas =                                                    1
  | F1 of tailleEntree * platPrincipal                          2
  | F2 of platPrincipal * fromOUdess                            3
```

Q7. (0,5pt) Donner les définitions en OCAML des constantes correspondant aux repas suivants :

a) une formule 1 composée d'une petite entrée et du plat principal 3,

b) une formule 2 composée du plat principal 1 et d'un morceau de fromage.

Q8. (0,5pt) Quel est le coût des repas de la question précédente ?

Q9. (1,5pt) Implémenter une fonction *coutRepas* qui calcule le coût d'un repas.

Étendons maintenant la notion de repas :

- $Formule_1$: une entrée et un plat,
- $Formule_2$: un plat et un dessert ou un fromage,
- $Formule_3$: une entrée, un plat, un fromage *et*/*ou* un dessert.

Q10. (2pt) Modéliser la notion « fromage et/ou dessert » par un type appelé `fromETOUdess`, puis implémenter le nouveau type `repas2`.

Q11. (2pt) Donner les modifications à apporter à la fonction *coutRepas* précédente pour calculer le coût d'un `repas2`.

3.2 Partiel 2013–2014 TD3 colorimétrie

En informatique, le terme *colorimétrie* regroupe les différents systèmes de gestion des couleurs des périphériques (écran, projecteur, imprimante, scanner, appareil photo, vidéo, ...).

Le but de cet exercice est d'implémenter le format de codage *RVB*, cet acronyme désignant les trois couleurs primaires [2] : rouge, vert et bleu.

3.2.1 Les couleurs

Dans le système colorimétrique RVB, on considère qu'une couleur est un mélange des trois couleurs – rouge, vert et bleu – selon une certaine intensité.

L'*intensité* est un entier naturel compris entre 0 (intensité minimale : couleur correspondante absente) et 255 (intensité maximale de la couleur correspondante).

On définit donc une couleur comme un triplet (r, v, b), où la composante r représente l'intensité du rouge, v l'intensité du vert et b celle du bleu.

Q1. (1pt) Compléter l'implémentation du type `tripletRVB` :

```
type intensite  = int     (*  restreint à  {0, ..., 255} *)    1
type tripletRVB = ..................................................  2
```

Les couleurs primaires ont une intensité maximale pour leur propre composante, minimale pour les autres ; le noir est défini [1] comme l'absence de couleur ; le blanc est défini [1] comme le mélange à intensité maximale des trois couleurs primaires.

[2] en synthèse additive

Q2. (1,25pt) Implémenter ces cinq couleurs grâce à la construction OCaml :
`let ... : tripletRVB = ...`

3.2.2 Le gris

Un triplet RVB est dit *gris* quand l'intensité de ses trois composantes est la même.

Q3. (1,5pt) Définir (spécification + réalisation) une fonction appelée *estGris* qui teste si un triplet RVB donné quelconque est gris. Dans la section « exemple » de la spécification, on veillera à donner un exemple retournant vrai et un exemple retournant faux.

Le *niveau de noir* d'un triplet RVB gris est le pourcentage de noir présent dans ce triplet. Par exemple, le niveau de noir du blanc est 0, tandis que le niveau de noir du noir est 100.

Q4. (1,25pt) Définir (spécification + réalisation) une fonction appelée *niveauNoir* qui donne le niveau de noir d'un triplet RVB (gris) quelconque, sous forme d'un réel entre 0 et 100.

3.2.3 Barbouillons

Afin de pouvoir mélanger les couleurs, on spécifie la fonction suivante :

SPÉCIFICATION 2	
PROFIL	$barbouille : tripletRVB \to tripletRVB \to tripletRVB$
SÉMANTIQUE	$(barbouille\ t_1\ t_2)$ est la couleur obtenue en mélangeant t_1 et t_2
EX. ET PROP.	(i) soit jaune $= (255, 255, 0)$, $(barbouille$ rouge vert $) = (barbouille$ vert rouge $) =$ jaune , (ii) $(barbouille$ bleu jaune $) = (barbouille$ jaune bleu $) =$ blanc

Q5. (1,5pt) En se basant sur l'algorithme suivant, donner l'implémentation de cette fonction.

ALGORITHME Les intensités sont additionnées composante par composante. Si le résultat de cette addition dépasse 255, l'intensité correspondante est fixée à 255.

3.2.4 Du gris et des couleurs

On souhaite maintenant manipuler divers modèles de gestion de couleurs, en se basant sur la définition suivante :

$$couleur \stackrel{def}{=} \{Rouge, Vert, Bleu\} \cup \{Noir(p) \ / \ p \in [0,100]\} \cup \{Rvb(t) \ / \ t \in tripletRVB\}$$

Dans cette définition, p est un réel compris entre 0 et 100.

Q6. (1,75pt)

a) Quelle est la nature de *Rouge*, *Vert*, *Bleu*, *Noir* et *Rvb* dans la définition ci-dessus ?

b) Donner une implémentation en OCaml de l'ensemble *couleur*.

Couleurs primaires

Les couleurs *primaires* sont le rouge pur, le vert pur, le bleu pur, et ce sont les seules.

Q7. (1,75pt) Implémenter en OCaml une fonction appelée *estPrimaire* qui teste si une couleur quelconque est primaire.

Couleurs complémentaires

Le *complément à x* d'un nombre est la quantité qui lui manque pour « aller à » x. Par exemple, le complément à 10 de 7 est 3, car $7 + 3 = 10$.

Le *complémentaire* d'une couleur est défini par les règles suivantes :

- le complémentaire du rouge est le cyan (intensités RVB = 0, 255, 255), le complémentaire du vert est le magenta (intensités = 255, 0, 255), le complémentaire du bleu est le jaune (255, 255, 0) ;
- le complémentaire d'un noir de niveau n est le noir dont le niveau est le complément à 100 de n ;
- le complémentaire d'une couleur dont les intensités RVB sont r, v et b est la couleur dont les intensités sont les compléments à 255 de r, v et b.

Q8. (2pt) Implémenter en OCaml une fonction appelée *complementaire* qui retourne la couleur complémentaire d'une couleur donnée quelconque.

Barbouillons encore

On veut à nouveau mélanger deux couleurs, en réutilisant les objets déjà définis. Pour cela, on spécifie une fonction de conversion des couleurs vers les triplets RVB :

SPÉCIFICATION 3	
PROFIL	$couleurVtriplet : couleur \rightarrow tripletRVB$
SÉMANTIQUE	$couleurVtriplet(c)$ est le triplet RVB correspondant à c.

Q9. (1,5pt) Implémenter *couleurVtriplet* en OCaml.

Q10. (1,5pt) En déduire une implémentation de la fonction de mélange des couleurs, spécifiée ainsi :

PROFIL $barbouilleC : couleur \rightarrow couleur \rightarrow couleur$

SÉMANTIQUE $(barbouilleC\ c_1\ c_2)$ est la couleur obtenue en mélangeant c_1 et c_2

Deuxième partie

DÉFINITIONS RÉCURSIVES

Chapitre 4

Rappels

Vocabulaire

- *Définir* = donner une définition ; *définition* = spécification + réalisation
- *Spécifier* = donner une spécification (le « quoi ») ; *pécification* = profil + sémantique + examples et/ou propriétés
- *Réaliser* = donner une réalisation (le « comment ») ; *réalisation* = algorithme (langue naturelle et/ou équations récursives + terminaison) + implémentation (OCaml)
- *Implémenter* = donner une inplémentation (OCaml)

Dans certains cas, certaines de ces rubriques peuvent être omises.

Chapitre 5

Fonctions récursives sur les entiers

Sommaire

Dans le titre de chaque exercice, problème ou annale d'examen est indiqué TDn ou TPn , où n est un entier entre 1 et 12 correspondant au numéro de séance de TD ou TP **à partir duquel** l'énoncé peut être abordé. En pratique, un exercice n^o n sera peut-être traité en séance $n+1$ ou $n+2$, et les exercices ne sont pas forcément traités dans l'ordre du polycopié.

Il y a plus d'énoncés que de séances ; ne pas hésiter à s'entraîner sur les énoncés qui n'auront pas été abordés en séance, notamment pendant le 2^e créneau de TP (non supervisé).

5.1 TD4 Additions des entiers de Peano

On souhaite additionner deux entiers de Peano, dont on rappelle une définition :

$$natP \stackrel{def}{=} \{Z\} \cup \{S(p) \text{ tel que } p \in natP\}$$

Q1. Quel est le *natP* correspondant à l'entier 0 ? À l'entier 2 ?

Q2. Implémenter *natP* en OCaml.

5.1.1 Addition dans *natP*

Q3. Compléter la spécification de la fonction d'addition de deux *natP* ci-dessous :

SPÉCIFICATION 1	
PROFIL	$addP$: ..... → ....
SÉMANTIQUE	$(addP\ n_1\ n_2)$ est l'entier de Peano représentant la somme de n_1 et n_2.
EX. ET PROP.	(i) $(addP\ (S\ (S\ Z))\ (S\ Z)) =$ .....
	(ii) $(addP$ .....$) = S\ (S\ (S\ Z))$
	(trouver un autre exemple que le précédent)
	(iii) ..... $= Z$
	(iv) $\forall n \in$ ...., $(addP\ n\ Z) =$.

Réalisation récursive de *addP*

Q4. En se basant sur la structure récursive du type du 1er paramètre de *addP*, compléter les équations récursives ci-dessous :

(1) $(addP\ \ldots.\ n_2) = \ldots$

(2) $(addP\ \ldots..\ \ldots) = \ldots\ldots\ldots\ldots$

Q5. Compléter la définition de la mesure ci-dessous (rappeler les propriétés qu'elle doit vérifier) et montrer que $\forall n_1,\ n_2 \in natP$, l'évaluation de $(addP\ n_1\ n_2)$ termine.

> **Indication** On rappelle que les paramètres de la *mesure* sont les mêmes (en nombre et en type) que la fonction dont on veut démontrer la terminaison (ici *addP*).
>
> Pour la définir, on pourra considérer que les *natP* sont des ensembles de S, et utiliser le cardinal (noté $|\,|$) qui donne le nombre d'éléments d'un ensemble.

$(mesure\ \ldots..) \stackrel{def}{=} \ldots$

⋮

Q6. Compléter la réalisation de *addP* ci-dessous :

RÉALISATION 1
Équations récursive et terminaison : voir questions précédentes
ALGORITHME analyse par cas par filtrage
IMPLÉMENT. ⋮

5.1.2 Conversion des *natP* en entiers naturels

Manipuler les entiers de Peano plutôt que la notation usuelle 0, 1, 2, ... est assez fastidieux ; on souhaite donc définir des fonctions de conversion des entiers de Peano vers les entiers naturels, et vice versa.

Q7. Implémenter $\mathbb{N}$ en OCaml ; on prendra soin de préciser toute restriction éventuelle.

Q8. Spécifier puis réaliser (équations, terminaison, implémentation) la fonction de conversion des *natP* vers les entiers naturels. On donnera quelques exemples significatifs.

Q9. Spécifier puis implémenter la fonction de conversion des entiers naturels vers les *natP* (réciproque de la précédente).

Q10. Déduire des questions précédentes une implémentation non récursive de *addP*.

Chapitre 6

Types récursifs

Sommaire

Dans le titre de chaque exercice, problème ou annale d'examen est indiqué TDn ou TPn , où n est un entier entre 1 et 12 correspondant au numéro de séance de TD ou TP **à partir duquel** l'énoncé peut être abordé. En pratique, un exercice n^o n sera peut-être traité en séance $n+1$ ou $n+2$, et les exercices ne sont pas forcément traités dans l'ordre du polycopié.

Il y a plus d'énoncés que de séances ; ne pas hésiter à s'entraîner sur les énoncés qui n'auront pas été abordés en séance, notamment pendant le 2^e créneau de TP (non supervisé).

6.1 TD4 Puissances d'entiers premiers

On souhaite représenter des entiers de la forme x^p tels que :

- x, appelé *racine*, est un entier premier > 1 ;
- p, appelé *exposant*, est un entier ≥ 1.

6.1.1 Définition des types

Trois représentations différentes définies par les ensembles *puis*, *puiscple* (un type produit) et *puisrec* (un type récursif) seront utilisées :

déf $puis = \mathbb{N}^* \backslash \{1\}$

déf $rac = \{x \in \mathbb{N}^* \backslash \{1\} \textit{ tel que } x \textit{ premier}\}$

déf $expo = N^*$

déf $puiscple = rac \times expo$

déf $puisrec = \{R(x) \textit{ tel que } x \in rac\} \cup \{P(p) \textit{ tel que } p \in puisrec\}$

Exemples de représentations dans les 3 types

expression	*puis*	*puiscple*	*puisrec*
3^1	3	(3,1)	$R\ 3$
3^3	27	(3,3)	$P\ (P\ (R\ 3))$

Le constructeur P représente une opération d'incrémentation de l'exposant. Le constructeur R injecte l'entier racine en une valeur de type *puisrec*.

Q1.

a) Donner les représentations des constantes suivantes dans les trois types de puissances : 5^1, 5^2, 7^3 et 23.

b) Implementer *puis*, *rac*, *expo*, *puiscple* et *puisrec*en OCaml.

c) Définir en OCaml les constantes correspondant aux puissances 5^1, 5^2, 7^3 et 23 dans les types *puis*, *puiscple* et *puisrec*.

6.1.2 Racine et l'exposant d'une valeur de type *puisrec*

Q2. Compléter la spécification des fonctions *racine* et *exposant* d'extraction des composantes :

SPÉCIFICATION 1	
PROFIL	$racine : puisrec \to rac$
SÉMANTIQUE	$racine(pr)$ est la racine de pr.
EX. ET PROP.	(i) $racine\ (\ R\ 3\ ) = \ldots$
	(ii) $racine\ (\ P\ (P\ (R\ 7))\ ) = \ldots$
PROFIL	$exposant : \ldots\ldots\ldots\ldots$
SÉMANTIQUE	$exposant(pr)$ est l'exposant de pr.
EX. ET PROP.	(i) $exposant\ (\ R\ 3\ ) = \ldots$
	(ii) $exposant\ (\ P\ (P\ (R\ 7))\ ) = \ldots$

Q3. En se basant sur la structure récursive du type *puisrec*, compléter les équations définissant les fonctions *racine* et *exposant* :

- Équations de récurrence :
  (1) $racine(R\ r) = \ldots$
  (2) $racine(P\ pr_2) = \ldots\ldots\ldots$
- Équations de récurrence :
  (1) $exposant(\ldots) = \ldots$
  (2) $exposant(\ldots\ldots) = \ldots\ldots\ldots\ldots\ldots$

Définissons une mesure c'est-à-dire une fonction (au sens mathématique) permettant de démontrer la terminaison d'une définition récursive de fonction (au sens informatique).

Q4.

- Que peut-on dire des arguments d'une fonction mesure ?
- Rappeler les propriétés que doit posséder une telle fonction.

Soit $pr \in puisrec$; on propose la définition suivante :

$$mesure(pr) \stackrel{def}{=} |pr|,$$

où pr est vu comme un ensemble de P ; $|\ |$ dénote donc ici un *cardinal*, c'est-à-dire un nombre d'éléments.

Q5. Montrer que $\forall pr \in puisrec$, l'évaluation de $racine(pr)$ termine.

Q6. Implementer *racine* et *exposant*.

6.1.3 Conversions d'un type vers un autre

Pour passer d'une représentation à une autre on spécifie des fonctions de conversion d'un type vers un autre :

SPÉCIFICATION 2	
PROFIL	$recVcple$: $puisrec \rightarrow puiscple$
SÉMANTIQUE	$recVcple(pr)$ est le $puiscple$ correspondant à pr.
EX. ET PROP.	(i) $recVcple(\ R\ 5\ ) = (5,1)$ (ii) $recVcple(\ P\ (P\ (R\ 11))\ ) = (11,3)$
PROFIL	$cpleVrec$: $puiscple \rightarrow puisrec$
SÉMANTIQUE	$cpleVrec(pc)$ est le $puisrec$ correspondant à pc.
EX. ET PROP.	(i) $cpleVrec(5,1) = R\ 5$ (ii) $cpleVrec(11,3) = P\ (P\ (R\ 11))$
PROFIL	$recVpuis$: $puisrec \rightarrow puis$
SÉMANTIQUE	$recVpuis(pr)$ est le $puis$ correspondant à pr.
EX. ET PROP.	(i) $puisrec(P\ (P\ (R\ 3))) = 27$
PROFIL	$cpleVpuis$: $puiscple \rightarrow puis$
SÉMANTIQUE	$cpleVpuis(pc)$ est le $puis$ correspondant à pc
EX. ET PROP.	(i) $puiscple(11,3) = 1331$

Q7. Implémenter *recVcple* en utilisant *racine* et *exposant*. Proposer une réalisation récursive plus efficace en termes de redondance des calculs.

Q8. Donner une réalisation récursive de *cpleVrec*.

Q9. Donner une réalisation récursive de *recVpuis*.

Q10. Donner une réalisation récursive de *cpleVpuis*.

6.1.4 Produit de deux puissances

On souhaite calculer le produit de deux puissances (telles qu'elles ont été définies au paragraphe 6.1.1 p. 33).

Comme on le verra dans la suite, ce produit n'est pas toujours défini ; dans un tel cas, on décide de renvoyer *None* («aucun» en Français), ce qui signifie qu'aucune valeur ne convient. Si le produit est défini et vaut v, on renvoit $Some(v)$ («une» en Français), ce qui signifie qu'une valeur - v - convient.

None et $Some(v)$ ont le même type : $option(\alpha)$, où α est le type de v. Ce sont les deux constructeurs de ce type, qui est prédéfini dans la librairie standard d'OCaml ($\geqslant 4.08$) par [1] :

```
type 'a option = None | Some of 'a
```

Utilisés conjointement, `None` et `Some` permettent donc de résoudre élégamment le problème des fonctions partielles.

[1] ocaml.org/api/Option.html

On spécifie donc :

SPÉCIFICATION 3	
PROFIL	$prod\_cple$: $puiscple \to puiscple \to option(puiscple)$
SÉMANTIQUE	$(prod\_cple\ x\ y)$ est le produit des deux puissances x et y. **Précondition** : x et y ont la même racine. Ce produit vaut *None* si la précondition n'est pas vérifiée.
PROFIL	$prod\_rec$: $puisrec \to puisrec \to option(puisrec)$
SÉMANTIQUE	$(prod\_rec\ x\ y)$ est le produit des deux puissances x et y. **Précondition** : x et y ont la même racine. Ce produit vaut *None* si la précondition n'est pas vérifiée.

Q11. Justifier la précondition.

Q12. Implémenter *prod_cple*.

Q13. Donner une réalisation de *prod_rec*.

6.2 TD5 Séquences construites par la gauche

Rappelons une définition (vue en cours) des séquences d'entiers construites par ajout à gauche d'un élément (un entier) :

$$seq \stackrel{def}{=} \{Nil\} \ \cup \ \{Cons(pr,fin) \ tel\ que\ pr \in \mathbb{Z}\ ,\ fin \in seq\}$$

Q1. Implémenter cette définirion de *seq* en OCaml.

Q2. Définir en OCaml les constantes cst_1 correspondant à la séquence ne comportant que l'élément 5 et cst_2 correspondant à la séquence formée des éléments 6, -3 et 7 (dans cet ordre).

6.2.1 Élément le plus à droite

Version 1

On veut définir une fonction qui donne l'entier le plus à droite d'une séquence d'entiers construite par la gauche, c'est-à-dire le dernier. On propose la spécification suivante :

SPÉCIFICATION 1 — INCORRECTE	
PROFIL	dernier : $seq \to \mathbb{Z}$
SÉMANTIQUE	$dernier(s)$ est le dernier élément de la séquence s

Q3. En quoi cette spécification est-elle incorrecte ?

Q4. En se basant sur la structure récursive de *seq*, donner les équations définissant la fonction *dernier* en respectant la contrainte sur son domaine.

Q5. Définir une mesure (rappeler les propriétés qu'elle doit vérifier) et démontrer que $\forall\ s \in seq \backslash \{Nil\}$, l'évaluation de $dernier(s)$ termine.

> **Indication** Pour définir la mesure, on pourra considérer que s est un ensemble de *Cons*, et utiliser le cardinal (noté $||$) qui donne le nombre d'élément d'un ensemble.

Q6. Implémenter *dernier* en OCaml. Quel message donnera l'interpréteur à l'issue de l'évaluation de cette implémentation ?

Version 2

Dans le cadre d'un développement plus complexe, l'utilisation de l'attribut `[@@warning "-8"]` en cas de `pattern-matching non exhaustive` est évidemment très dangereuse, et requiert une étude précise de la complétude des équations, ce qui est parfois difficile.

Pour éviter cet écueil, on peut utiliser le type `'a option`, prédéfini dans la librairie standard d'OCaml[2] et reproduit ci-dessous :

```
type 'a option = None | Some of 'a
```

`None` («aucun» en Français) signifie qu'aucune valeur ne convient ; `Some` («une» en Français) signifie qu'une valeur convient. De part son profil polymorphe, `Some` peut s'appliquer à n'importe quelle valeur.

On spécifie ainsi une 2e version pour le dernier entier d'une séquence :

SPÉCIFICATION 2	
PROFIL	$dernier_{v2} : seq \to option(\mathbb{Z})$
SÉMANTIQUE	Si l'élément le plus à droite de s n'est pas défini, $dernier_{v2}(s) = None$; sinon, soit `d` cet élément, $dernier_{v2}(s) = Some(d)$.

Remarquer que le domaine de $dernier_{v2}$ est maintenant seq : la valeur `Nil` n'est plus exclue.

Q7. Compléter la spécification de $dernier_{v2}$ par des exemples d'utilisation.

Q8. Réaliser (équations récursives, terminaison, implémentation) $dernier_{v2}$.

Version 3

On suit une démarche analogue à la 2e version, mais basée sur un couple $\mathbb{B} \times \mathbb{Z}$ à la place du type `'a option` :

SPÉCIFICATION 3	
PROFIL	$dernier_{v3} : seq \to \mathbb{B} \times \mathbb{Z}$
SÉMANTIQUE	Si l'élément le plus à droite de s n'est pas défini, $dernier_{v3}(s) = (faux, ??)$, où ?? est un entier quelconque ; sinon, soit `d` cet élément, $dernier_{v3}(s) = (vrai, d)$.

[2] ocaml.org/api/Option.html

EX. ET PROP. (i) $dernier_{v3}(Nil) = (faux, ??)$;?? représente n'importe quel entier
(ii) $dernier_{v3}(Cons(6, Nil)) = (vrai, 6)$
(iii) $dernier_{v3}(Cons(3, (Cons(5, Nil)))) = (vrai, 5)$

Q9. Définir (équations, terminaison, implémentation) $dernier_{v3}$.

6.2.2 Généralisation : séquence de type quelconque

On revient sur la 1re version de *dernier*.

Q10. Indiquer toutes les modifications à faire si on considère des séquences de caractères au lieu de séquences d'entiers.

6.3 TD5 Séquences construites par la droite

La construction des séquences par la gauche a été étudiée en cours et dans l'exercice 6.2 page 36. Dans cet exercice, on étudie une construction alternative : les éléments (des entiers) seront ajoutés par la droite.

6.3.1 Le type seq_d des séquences construites par la droite

On définit : $seq_d \stackrel{def}{=} \{Nil_d\} \cup \{Cons_d(déb, der) \text{ tel que } déb \in seq_d, der \in \mathbb{Z}\}$

Schématiquement, une seq_d s non vide peut être représentée par | ... déb ... | der | où déb $\in seq_d$ et der $\in \mathbb{Z}$; déb est le début de s (c'est une séquence), der est le dernier élément de s (c'est donc un entier).

Q1. Implémenter seq_d en OCaml.

Q2. Définir en OCaml les constantes $cstd_1$ correspondant à la séquence ne comportant que l'élément 5 et $cstd_2$ correspondant à la séquence formée des éléments 6, −3 et 7 (dans cet ordre).

6.3.2 Longueur d'une seq_d

Q3. Spécifier une fonction $long_d$ qui, étant donnée une séquence $s \in seq_d$, retourne le nombre d'éléments de s.

Q4. Donner des équations récursives définissant $long_d$.

Q5. Définir une mesure, et montrer que $\forall s \in seq_d$, l'évaluation de $(long_d\ s)$ termine.

Q6. Implémenter $long_d$.

6.3.3 Comparaison des seq_d avec les seq

La séquence $cstd_2$ a les mêmes éléments que la séquence cst_2 de l'exercice 6.2 – 6, -3, 7 – et ils sont énumérés dans le même ordre. De ce point de vue, ces deux séquences sont égales.

Q7. Que donnerait l'évaluation de `cst2 = cstd2` dans l'interpréteur OCaml ? Expliquer.

Une fonction de comparaison est donc nécessaire.

Q8. Spécifier une fonction de comparaison eg_{seq} telle que $\forall s_1 \in seq,\ \forall s_2 \in seq_d, (eg_{seq}\ s_1\ s_2)$ est vrai si et seulement si s_1 et s_2 ont les mêmes éléments énumérés dans le même ordre. On donnera trois exemples significatifs : (i) ...= vrai ; (ii) ... $= faux$ (mêmes éléments mais ordre différent) ; (iii) ... $= faux$ (éléments différents).

Pour réaliser eg_{seq}, on spécifie deux fonctions auxiliaires :

SPÉCIFICATION 3	
PROFIL	$prem\ :\ seq_d \setminus \{\mathbf{Nild}\} \rightarrow \mathbb{Z}$
SÉMANTIQUE	$(prem\ s)$ est le premier élément de la séquence s.
EX. ET PROP.	(i) $(prem\ cstd_1) = 5$ (ii) $(prem\ cstd_2) = 6$

SPÉCIFICATION 4	
PROFIL	$suite\ :\ seq_d \setminus \{\mathbf{Nild}\} \rightarrow seq_d$
SÉMANTIQUE	$(suite\ s)$ est la même séquence que s mais sans son premier élément.
EX. ET PROP.	(i) $(suite\ cstd_1) = Nd$ (ii) $(suite\ cstd_2) = Cd\ (Cd\ (Nd, -3), 7)$

Remarquer qu'étant donné que le premier élément d'une séquence vide n'existe pas, ni *prem*, ni *suite* ne sont définies sur la séquence vide *Nild*.

Q9. Réaliser *prem* (équations, terminaison, implémentation).

Q10. Implémenter *suite*.

Q11. Déduire des questions précédentes une réalisation de eg_{seq} (équations, terminaison, implémentation).

Q12. Donner une expression OCaml permettant d'établir que $cst_2 = cstd_2$

6.4 TD6 Séquences en OCaml

En OCaml, plusieurs implémentations des séquences existent (voir paragraphe 6.4.1 pour plus d'informations).

Précédemment, en cours et en TD (cf exercice 6.2 page 38 pour les séquences d'entiers), les séquences d'éléments de type α ont été implémentées par un type somme récursif, dont on rappelle une définition mathématique, suivie de son implémentation :

$$séq\ (\alpha) \stackrel{def}{=} \{Nil\} \cup \{Cons(pr, fin)\ tel\ que\ pr \in \alpha, fin \in séq\ (\alpha)\}$$

```
type 'a seq = Nil | Cons of 'a * 'a seq
```

Pour des raisons de *performance* (complexité algorithmique temporelle et spatiale), OCaml implémente les séquences en interne dans un langage de bas niveau (c'est-à-dire proche du processeur de la machine), le langage C[3] .

Pour des raisons de *praticité*, OCaml fournit du sucre syntaxique pour noter les séquences. Ainsi, tout se passe comme si elles étaient définies par le type somme récursif suivant :

[3] https://fr.wikipedia.org/wiki/C_(langage)

```
type 'a list = [] | (::) of 'a * 'a list
```

Q1. Compléter le tableau ci-dessous illustrant les correspondances entre les différentes notations des séquences :

α	**seq**	**list**
$\forall\, \alpha$	`Nil`	`..`
$\alpha =$.	`.....`	`(::)(5, [])`
$\alpha =$.	`.....`	`5 :: []`
$\alpha =$.	`.....`	`[5]`
$\alpha =$.	`Cons(5, Cons(0, Nil))`	`.....`
$\alpha =$.	`Cons(5, Cons(0, Nil))`	`.....`
$\alpha =$.	`Cons(5, Cons(0, Nil))`	`.....`
$\alpha =$ ..	`.....`	`[3.;.14]`
$\alpha =$ `.....`	`Cons((4,'4'), Cons((2,'2'), Nil))`	`.....`
$\forall\, \alpha$	Soit `pr:'a` et `fin : 'a seq` `Cons(pr, fin)`	`.....` `.....`

On remarquera que `::` est associatif à droite.

Q2. En s'inspirant de l'exercice 6.2, définir la fonction *dernier* (spécification, équations récursives, terminaison, implémentation).

6.4.1 Pour aller plus loin : séquences en OCaml

Pour information, voici une liste (non exhaustive) de diverses implémentations des séquences en OCaml :

- listes [4]
- listes paresseuses [5]
- tableaux [6]
- dictionnaires [7]
- queues [8] (FIFO [9])

[4] https://ocaml.org/api/List.html
[5] https://ocaml.org/api/Seq.html
[6] https://ocaml.org/api/Array.html
[7] https://ocaml.org/api/Map.html
[8] https://ocaml.org/api/Queue.html
[9] First-In, First-Out

- piles[10] (LIFO[11])
- ...

Dans ce cours, on étudie uniquement les listes, qui sont fondamentales pour l'algorithmique, notamment fonctionnelle.

Étant donné une séquence non vide s, les fonctions donnant respectivement le premier élément de s et s privée de son premier élément sont prédéfinies en OCaml dans le module `List` de la librairie standard[12] :

- `List.hd` : $séq\,(\alpha)\backslash\{[\,]\} \to \alpha$
- `List.tl` : $séq\,(\alpha)\backslash\{[\,]\} \to séq\,(\alpha)$

La concaténation de deux séquences également : `List.append` : $séq\,(\alpha) \to séq\,(\alpha) \to séq\,(\alpha)$. OCaml fournit du sucre syntaxique pour `append` via l'opérateur infixe `@`.

Q3. Compléter le tableau ci-dessous.

N°	EXPRESSION OCAML	ENSEMBLE / TYPE ensemble ↔ type OCAML	EVALUATION OCAML
1	$[3;-1;0]$	..... ↔	.....
2	$-2 :: [3;-1;0]$	..... ↔	.....
3	$3.14 :: [\,]$	..... ↔	.....
4	`'0' :: [ ]`	..... ↔	.....
5	$[3] :: -1$	.....	.....
6	$[-2;3]\,@\,[-1;0]$	..... ↔	.....
7	$-2\,@\,[3;-1;0]$	.....	.....
8	$[\,]\,@\,[1.2]$	..... ↔	.....
9	$[1.0;2]$	.....	.....
10	`['L']@['1']`	..... ↔	.....
11	`[12]@['3']`	.....	.....
12	*hd* $[-2;3;0]$	. ↔	...
13	*tl* $[-2;3;0]$	..... ↔	.....
14	*hd* $[3]$	. ↔	.
15	*tl* $[3]$	..... ↔	..
16	*hd* $[\,]$	.....	.....
17	*hd* $'a'$	.....	.....
18	*tl* $['a';'b'] = [\,]$	.. ↔	.....
19	$['F';'i']\,@\,....\,@\,['!']$	..... ↔	`['F';'i';'n';'!']`

[10] https://ocaml.org/api/Stack.html
[11] Last-In, First-Out
[12] https://ocaml.org/api/List.html

Comme les éléments d'une séquence sont de type quelconque (mais fixé : α), on peut imbriquer cette structure et former des séquences de séquences.

Q4. Compléter le tableau ci-dessous.

N°	EXPRESSION OCAML	ENSEMBLE / TYPE ensemble ↔ type OCAML	EVALUATION OCAML
1	$hd(tl\ [[4];[5];[6]])$	..... ↔	...
2	$(0, \texttt{'0'}) :: [(1,'1');(2,'2')]$	..... ↔	.....
3	$(1, true) :: [5]$	..... ↔	.....
4	$[(1,2);(3,4)]\,@\,[(5,6,7)]$	..... ↔	.....
5	$[[1;2];[3;4]]\,@\,[[5;6;7]]$	..... ↔	.....
6	$[[1;2];[3;4]]\,@\,[5;6;7]$	..... ↔	.....

Chapitre 7

Fonctions récursives sur les séquences

Sommaire

Dans le titre de chaque exercice, problème ou annale d'examen est indiqué TDn ou TPn , où n est un entier entre 1 et 12 correspondant au numéro de séance de TD ou TP **à partir duquel** l'énoncé peut être abordé. En pratique, un exercice n^o n sera peut-être traité en séance $n+1$ ou $n+2$, et les exercices ne sont pas forcément traités dans l'ordre du polycopié.

Il y a plus d'énoncés que de séances ; ne pas hésiter à s'entraîner sur les énoncés qui n'auront pas été abordés en séance, notamment pendant le 2^e créneau de TP (non supervisé).

7.1 TD6 Présence d'un élément

On cherche à savoir si un élément appartient à une séquence.

Q1. Implémenter l'ensemble *séq* (*caractère*), appelé *texte*.

La fonction *app*est spécifiée comme suit :

SPÉCIFICATION 1	
PROFIL	app : *caractère* $\rightarrow$ *texte* $\rightarrow \mathbb{B}$
SÉMANTIQUE	(app c t) vaut *vrai* si et seulement si $c \in t$

Q2. En se basant sur la structure récursive du type *texte*, donner des équations récursives définissant la fonction *app*.

Q3. Définir une mesure (rappeler les propriétés qu'elle doit vérifier) et démontrer que $\forall c \in$ *caractère*, $\forall t \in$ *texte*, l'évaluation de (*app c t*) termine.

> **Indication** Pour définir la mesure, on pourra considérer que t est un ensemble de `::`, et utiliser le cardinal (noté `||`) qui donne le nombre d'élément d'un ensemble.

Q4. En suivant l'algorithme précisé dans la réalisation 1 version n° 1 ci-dessous, implémenter *app*récursivement :

RÉALISATION 1 — version n° 1	
ALGORITHME	filtrage et composition conditionnelle
IMPLÉMENT.	⋮

Q5. En suivant l'algorithme précisé dans la réalisation 1 version n° 2 ci-dessous, implémenter *app*récursivement :

RÉALISATION 1 — version n° 2	
ALGORITHME	filtrage et composition booléenne (`if then else` interdit)
IMPLÉMENT.	⋮

Q6. En suivant l'algorithme précisé dans la réalisation 1 version n° 3 ci-dessous, implémenter *app*récursivement :

RÉALISATION 1 — version n° 3	
ALGORITHME	(⋆) **uniquement** des expressions booléennes (`if then else` et `match` interdits)
IMPLÉMENT.	⋮

> **Indication** On pourra utiliser des fonctions prédéfinies dans le module `List` de la librairie standard, sauf bien sûr `List.mem`.

Q7. Que faut-il modifier dans les réponses aux questions précédentes pour que *app* puisse s'appliquer sur n'importe quel type ? Par exemple :

- `(app 2 [1;3;2])`
- `(app 'm' ['e'; 't'; 'h'])`
- `(app (1,2) [(2,3);(1,2);(2,1)])`

Q8. Implémenter une fonction qui teste la présence de la lettre `'e'` dans un texte, sans refaire ce qui a déjà été fait dans les questions précédentes.

7.2 TD6 Nombre d'occurrences

On souhaite définir une fonction *nbOcc* qui calcule le nombre d'occurrences d'un élément donné dans une séquence. Le type des éléments de la séquence est quelconque.

Q1. Spécifier *nbOcc* ; on donnera des exemples significatifs (0, 1 ou 2 occurrences).

Q2. Réaliser la fonction *nbOcc* (équations récursives, terminaison, implémentation).

Q3. Implémenter une fonction non récursive qui calcule le nombre de `'a'` d'une séquence de lettres.

On considère la fonction suivante :

SPÉCIFICATION 2	
PROFIL	$pdAE$: *séq* (*lettre*) $\rightarrow \mathbb{B}$
SÉMANTIQUE	$pdAE(t) = vrai$ si et seulement si le texte t comporte strictement plus de caractères `'a'` que de caractères `'e'`

Q4. Compléter les exemples suivants :

(i) $pdAE$(`['e'; 'l'; 'e'; 'a'; 'n'; 'o'; 'r']`) = ...

(ii) $pdAE$(`['N'; 'a'; 't'; 'h'; 'a'; 'l'; 'i'; 'e']`) = ...

Q5. Compléter la réalisation de *pdAE*ci-dessous (sans réécrire du code déjà écrit). On pourra définir une fonction intermédiaire si nécessaire.

RÉALISATION 2	
ALGORITHME	composition fonctionnelle, utilisation de `if then else` interdite
IMPLÉMENT.	⋮

La trace (`#trace` dans l'interpréteur OCaml) simplifiée de l'évaluation de `pdAE ['e'; 'l'; 'e'; 'a']` donne :

```
pdAE <-- ['e'; 'l'; 'e'; 'a']
nbOcc <-- ['e'; 'l'; 'e'; 'a']
nbOcc <-- ['l'; 'e'; 'a']
nbOcc <-- ['e'; 'a']
nbOcc <-- ['a']
nbOcc <-- []
nbOcc --> 0
nbOcc --> 0
nbOcc --> 1
nbOcc --> 1
nbOcc --> 2
nbOcc <-- ['e'; 'l'; 'e'; 'a']
nbOcc <-- ['l'; 'e'; 'a']
nbOcc <-- ['e'; 'a']
nbOcc <-- ['a']
nbOcc <-- []
nbOcc --> 0
nbOcc --> 1
nbOcc --> 1
nbOcc --> 1
nbOcc --> 1
pdAE --> false
```

Q6. Indenter cette trace de façon à faire apparaître l'enchaînement des appels récursifs. Combien de fois la séquence `['e'; 'l'; 'e'; 'a']` est-elle parcourue ?

On constate donc que les calculs sont redondants. Pour palier ce gaspillage de ressources, on spécifie la fonction suivante :

SPÉCIFICATION 3	
PROFIL	$nbAE$: *séq* (*caractère*) $\to \mathbb{N} \times \mathbb{N}$
SÉMANTIQUE	Posons $(n_a, n_e) = nbAE(t)$; n_a est le nombre de `'a'` et n_e le nombre de `'e'` de t.

Q7. Implémenter *nbAE*récursivement et sans utiliser de fonction intermédiaire.

Q8. Implémenter *pdAE*. On prendra soin de ne parcourir qu'une fois la séquence paramètre.

Q9. Donner la trace identée de l'évaluation de `['e'; 'l'; 'e'; 'a']`.

Après réflexion, on remarque que calculer le nombre d'occurrence de `'a'` et de `'e'` est finalement inutile : il suffit en effet de calculer la différence de ces nombres grâce à une fonction *dif AE* :

SPÉCIFICATION 4	
PROFIL	$difAE$: *séq* (*lettre*) $\to \mathbb{Z}$
SÉMANTIQUE	$difAE(t)$ est la différence entre le nombre de `'a'`et le nombre de `'e'` dans le texte t.

Q10. Implémenter *dif AE*, puis ré-implémenter *pdAE*en utilisant *dif AE*.

On laissera le lecteur se convaincre que la trace est la même que dans la question précédente.

7.3 TD6 Maximum d'une séquence

On s'intéresse à l'élément maximum d'une séquence :

SPÉCIFICATION 1	
PROFIL	$maxN$: *séq* $(\alpha)\backslash\{[\,]\} \to \alpha$
SÉMANTIQUE	$maxN(s)$ est le plus grand (au sens de $\geqslant$) des éléments de s.

Q1. Compléter les exemples suivants :

(i) $maxN$`(['z'])` = ...

(ii) $maxN$`([-.3.;-.0.14])` = ...

(iii) $maxN$`([2;3;1;3;1])` = .

(iv) $maxN$`([])` = ...

Q2. Donner des équations récursives définissant *maxN*. On pourra utiliser la fonction suivante :

SPÉCIFICATION 2	
PROFIL	$max : \alpha \to \alpha \to \alpha$
SÉMANTIQUE	$(max\ x\ y)$ est la valeur maximum $(\geqslant)$ entre x et y.

prédéfinie en OCaml.

Q3. Définir une mesure (rappeler les propriétés qu'elle doit vérifier) et démontrer que $\forall s \in séq\ (\alpha)\backslash\{[\,]\}$, l'évaluation de $maxN(s)$ termine.

> **Rappel.** Pour définir la mesure, on pourra considérer que s est un ensemble de `::`, et utiliser le cardinal (noté `||`) qui donne le nombre d'élément d'un ensemble.

Q4. Implémenter *maxN* en OCaml.

On définit de manière analogue la fonction :

SPÉCIFICATION 3	
PROFIL	$minN : séq\ (\alpha)\backslash\{[\,]\} \to \alpha$
SÉMANTIQUE	$minN(s)$ est le plus petit (au sens de $\leqslant$) des éléments de s.

La réalisation de *minN* n'est pas demandée.

On souhaite définir maintenant une fonction qui calcule conjointement le plus grand et le plus petit élément d'une séquence non vide :

SPÉCIFICATION 4	
PROFIL	$minmax : séq\ (\alpha)\backslash\{[\,]\} \to \alpha \times \alpha$
SÉMANTIQUE	Posons $(mini, maxi) = minmax(s)$; $mini$ (resp. $maxi$) est le minimum (resp. maximum) de s.
EX. ET PROP.	(i) $minmax([4]) = (4,4)$ (ii) $minmax([3;5;7;1;8;3]) = (1,7)$

Q5. Déduire des questions précédentes une implémentation directe (non récursive) de la fonction *minmax*. On prendra soin de préciser les restrictions éventuelles.

Q6. Qu'observerait-on en traçant l'évaluation de `minmax([5;7;1])` ?

Q7. `(*)` Implémenter récursivement la fonction *minmax* sans utiliser les fonctions *maxN* et *minN* et en ne réalisant qu'un seul parcours de la séquence donnée en paramètre.

Q8. Qu'observerait-on maintenant en traçant l'évaluation de `minmax([5;7;1])` ?

7.4 TD6 Suppression de caractères d'un texte

On considère des *textes*, sous forme de séquences de caractères.

Q1. Définir une fonction *suppr* qui, étant donné un caractère c et un texte t, supprime toutes les occurrences de c dans t. Les autres caractères, ainsi que leur ordre d'apparition dans le texte, sont préservés.

Q2. Que faut-il modifier dans la définition précédente pour ne supprimer *qu'au début* du texte ? Par exemple : (*supprDeb* `'1' ['1';'0';'1']`) = `['0';'1']`

Q3. Que faut-il modifier dans les définitions précédentes pour généraliser le type des éléments de la séquence ?

7.5 TD7 Jouons aux cartes

L'objectif de cet exercice est d'expérimenter la définition d'une fonction récursive (partie 2 du cours) sur une hiérarchie complexe de types (révision de la partie 1).

On considère un jeu de 54 cartes (52 cartes et 2 jokers) modélisé comme suit :

- $joker \stackrel{def}{=} \{\text{JokRouge}, \text{JokNoir}\}$
- $enseigne \stackrel{def}{=} \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$
- $basse \stackrel{def}{=} 2, \ldots, 10$
- $petite \stackrel{def}{=} basse \times enseigne$
- $haute \stackrel{def}{=} \{\text{As}, \text{Roi}, \text{Dame}, \text{Valet}\}$
- $honneur \stackrel{def}{=} haute \times enseigne$
- $carte \stackrel{def}{=} \{\text{H}(h) \ / \ h \in honneur\} \cup \{\text{P}(p) \ / \ p \in petite\} \cup \{\text{J}(j) \ / \ j \in joker\}$

Q1. Préciser la nature (énuméré, somme, produit, intervalle, synonyme) de chacun de ces types (plusieurs réponses possibles), et en donner une réalisation en OCaml.

7.5.1 Valeur d'un joker

Dans ce jeu, imaginaire, un joker vaut 50 points quelle que soit sa couleur.

Q2. Donner la définition de la constante valJoker.

7.5.2 Valeur d'une petite

La valeur d'une petite est sa valeur faciale, indépendamment de sa couleur. Par exemple, le 9 de carreau vaut 9 points, ainsi que le 9 de pique : $valPetite(9, \diamondsuit) = 9$, $valPetite(9, \spadesuit) = 9$.

Q3. Définir *valPetite*.

7.5.3 Valeur d'un honneur

La valeur d'un honneur est également indépendante de sa couleur : c'est respectivement 11, 12, 13, 14 pour un valet, une dame, un roi, un as.

Q4. Spécifier *valHonneur* (on donnera des exemples d'utilisation), puis en donner une réalisation.

7.5.4 Valeur d'une main

La *main* est l'ensemble des cartes tenues par le joueur.

Q5. Proposer une définition du type *main*.

Q6. Définir la main *cstM*1 comprenant une dame de trèfle, un joker rouge et un neuf de carreau.

La fonction *valMain* calcule la somme des valeurs des cartes de la main.

Q7. Que vaut *valMain*(*cstM*1) ? Donner une spécification de *valMain* puis des équations récursive la définissant et enfin une réalisation.

7.6 TD7 Ordre alphabétique

On considère des *mots*, sous forme de séquences de caractères. On souhaite comparer des mots selon l'ordre lexicographique (généralisation de l'ordre alphabétique).

Q1. Définir le type mot.

Q2. Définir une fonction *egmot* déterminant si deux mots sont égaux.

Q3. Quel opérateur OCaml prédéfinit (et généralise) *egmot* ? Rappeler son profil.

Q4. Que faut-il modifier dans la réponse à la question 2 pour définir une fonction *inf mot* déterminant si un mot est avant un autre (strictement), selon l'ordre lexicographique ?

Q5. Quel opérateur OCaml prédéfinit (et généralise) *inf mot* ?

7.7 TD7 Valeurs cumulées

On souhaite étudier une fonction *valcum* qui calcule les *valeurs cumulées* d'une séquence d'entiers :

SÉMANTIQUE Soit $n \in \mathbb{N}^*$,
$valcum\ [e_1\ ;\ e_2\ ;\ e_3\ ;\ldots;\ e_n] = [e_1\ ;\ e_1+e_2\ ;\ e_1+e_2+e_3\ ;\ \ldots\ ;\ e_1+\ldots+e_n]$

Selon la façon de construire les séquences (par ajout des entiers à gauche ou bien à droite) et de les découper (à gauche ou bien à droite), quatre méthodes de réalisation sont possibles.

7.7.1 (*) Séquences construites et découpées à droite

On rappelle la définition des séquences construites à droite, étudiée précédemment dans l'exercice 6.3 p. 38 :

```
type seqd = NilD | ConsD of seqd*int
```

Pour alléger les notations, on utilisera le schéma suivant :

$$\boxed{e_1\ \ldots\ e_{n-1}\mid e_n} \stackrel{def}{=} ConsD(\ldots ConsD(NilD,\ e_1),\ldots,\ e_n)$$

À droite de |, il y a toujours un seul entier ; à gauche, il y a toujours une séquence (même si celle-ci est vide ou ne comporte qu'un entier).

Q1. Spécifier *valcum*. On donnera deux exemples d'utilisation : un avec une séquence singleton, l'autre avec la séquence $\boxed{5\,4\,3\,2\,0\mid 1}$.

SPÉCIFICATION 1	
PROFIL	$valcum$: $seqd \to seqd$
EX. ET PROP.	(i) $valcum(\boxed{\mid 5}) = \boxed{\mid 5}$
	(ii) $valcum(\boxed{5\,4\,3\,2\,0 \mid 1}) = \boxed{5\,9\,12\,14\,14 \mid 15}$

Q2.

1. Que vaut $valcum(\boxed{5\,4\,3\,2 \mid 0})$?
2. Quelles opérations doit-on effectuer pour passer du résultat de $valcum(\boxed{5\,4\,3\,2 \mid 0})$ à celui de $valcum(\boxed{5\,4\,3\,2\,0 \mid 1})$?
3. Définir une fonction qui serait utile pour calculer le dernier élément de la séquence des valeurs cumulées (sans calculer celle-ci pour l'instant).

Q3. Compléter les équations définissant $valcum$:

(1) $valcum(NilD) = \ldots\ldots$

(2) $valcum(ConsD(déb, der)) = \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$

7.7.2 (**) Séquences construites à gauche et découpées à droite

Le principe du découpage à droite précédent peut être appliqué à des séquences construites classiquement à gauche, en utilisant les fonctions suivantes :

SPÉCIFICATION 3	
PROFIL	$(@)$: $séq\,(\alpha) \to séq\,(\alpha) \to séq\,(\alpha)$
SÉMANTIQUE	concaténation des séquences
EX. ET PROP.	(i) $[1;2]@[3;4] = [1;2;3;4]$
PROFIL	$dernier$: $séq\,(\alpha)\backslash\{[\,]\} \to \alpha$
SÉMANTIQUE	dernier élément
EX. ET PROP.	(i) $dernier([1;2;3]) = 3$
PROFIL	$debut$: $séq\,(\alpha)\backslash\{[\,]\} \to séq\,(\alpha)$
SÉMANTIQUE	$debut(s)$ est la séquence obtenue en supprimant $dernier(s)$ dans s
EX. ET PROP.	(i) $debut([1;2;3]) = [1;2]$

Q4. Réécrire les équations de *valcum* du paragraphe précédent (dans les *seqd*) dans $séq\,(\alpha)$.

Indication Quelle relation peut-on écrire entre toute séquence *non vide* s, $@$, *debut*et *dernier* ?

Q5. Implémenter *valcum* ; l'implémentation de *debut*, *dernier* et *somme* ne sont pas demandées.

L'implémentation précédente est peu efficace, notamment à cause de l'utilisation conjointe de *debut* et *dernier*. Cela entraîne deux parcours de la séquence paramètre et donc une redondance des

appels récursifs. Il est parfaitement possible de calculer à la fois le début et le dernier en un seul parcours :

SPÉCIFICATION 4	
PROFIL	$debder : séq(\alpha)\backslash\{[\,]\} \to séq(\alpha) \times \alpha$
SÉMANTIQUE	Posons $(deb, der) = debder(s)$; deb est le début de s, der son dernier élément.

Q6. Compléter les exemples d'utilisation de *debder* :

(i) $(debder([1;2;3])) = \ldots\ldots\ldots$

(ii) $(debder([1])) = \ldots\ldots$

Q7. Donner des équations récursive définissant *debder*, et montrer que $\forall s \in séq(\alpha)\backslash\{[\,]\}$, l'évaluation de $debder(s)$ termine.

Indication Il faut combiner les équations de *debut* et *dernier* :

(1) $debut([pr]) = [\,]$
(2) $debut(pr :: fin) = pr :: (debut\ fin)$

(1) $dernier([pr]) = pr$
(2) $dernier(pr :: fin) = dernier(fin)$

Q8. Implémenter *debder*.

Q9. Que faut-il modifier dans l'implémentation de *valcum* afin d'utiliser *debder* à la place de *debut* et *dernier* ?

Q10. Implémenter *debut* et *dernier* grâce à *debder*.

7.7.3 (∗) Séquences construites et découpées à gauche

Q11.

1. Que vaut $valcum([4;3;2;0;1])$?
2. Quelles opérations doit-on effectuer pour passer du résultat de $valcum([4;3;2;0;1])$ à celui de $valcum([5;4;3;2;0;1])$?
3. Définir une fonction auxiliaire qui serait utile pour calculer la séquence des valeurs cumulées (sans calculer celle-ci pour l'instant).

Q12. En déduire des équations récursives définissant *valcum*.

7.7.4 (∗) Séquences construites et découpées (deux fois) à gauche

On souhaite maintenant donner une réalisation directe de *valcum*, c'est-à-dire n'utilisant pas de fonction intermédiaire. Pour cela, on étudie un découpage selon les deux premiers éléments.

Q13.

1. Quelle est la valeur de $valcum([1;3;5;7])$?
2. Quelle est la valeur de $valcum([4;5;7])$?
3. Remarquons que $4 = 1 + 3$. Comment passe-t-on de la valeur de $valcum([4;5;7])$ à la valeur de $valcum([1;3;5;7])$?

Q14. Compléter les équations suivantes :

(1) $valcum([\,]) = ..$

(2) $valcum([pr]) = ...$

(3) $valcum(pr_1 :: pr_2 :: suite) = ...$

7.8 TD7 (***) Le compte est bon

On étudie une forme simplifiée du jeu « le compte est bon »[1]. Étant donné un entier *cmpt* strictement positif et une séquence s d'entiers strictement positifs, on cherche à exprimer *cmpt* comme une somme d'éléments de s.

Par exemple, pour $cmpt = 20$ et $s = [18;3;1;4;20]$, on a une solution : 20. Pour $cmpt = 25$ et $s = [21;3;2;4;2]$, on a deux solutions : $21 + 4$ et $21 + 2 + 2$. Pour *cmpt* et s donnés, il peut ne pas y avoir de solution.

On ne fait aucune hypothèse sur la séquence donnée, elle peut être en ordre quelconque et comporter des répétitions. Il se peut ainsi que deux solutions soient identiques et on ne cherchera pas à optimiser cet aspect.

On appelle *solution* toute séquence d'entiers dont la somme des éléments vaut *cmpt*.

Indication : On pourra raisonner en suivant le principe suivant. Étant donné un compte *cmpt* à faire à partir d'une séquence $pr :: fin$, on considère les trois cas possibles :

a) $pr = cmpt$: alors le singleton $[pr]$ est une solution (il y en a peut-être d'autres).

b) $pr > cmpt$: alors pr n'appartient à aucune solution ; les solutions sont celles du « compte est bon » pour *cmpt* et fin.

c) $pr < cmpt$: alors les solutions comportant pr sont déduites de celles du « compte est bon » pour $cmpt - pr$ et fin (mais là encore, il y en a peut-être d'autres).

Pour commencer, on cherche juste à savoir s'il existe une solution :

SPÉCIFICATION 1	
PROFIL	$existeSol : \mathbb{N} \to séq(\mathbb{N}) \to \mathbb{B}$
SÉMANTIQUE	$(existeSol\ cmpt\ s) = vrai$ si et seulement si il existe au moins une solution au « compte est bon » avec *cmpt* et s.

[1] Le solveur de dcode.fr permet d'y jouer : www.dcode.fr/compte-est-bon

EX. ET PROP. (i) $(existeSol\ 20\ [18;3;1;4;20]) = vrai$
(ii) $(existeSol\ 25\ [21;3;2;4;2]) = vrai$
(iii) $(existeSol\ 1\ [21;3;2;4;2]) = faux$

Q1. Compléter les équations récursives définissant *existeSol* ci-dessous :

(1) $(existeSol\ cmpt\ [\,]) = \ldots.$

(2) $(existeSol\ cmpt\ pr::fin) =$
$\begin{cases} si\ pr = cmpt\ :\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ si\ pr > cmpt\ :\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ si\ pr < cmpt\ :\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \end{cases}$

Q2. En déduire une implémentation de *existeSol*.

Q3. Modifier les équations précédentes pour définir :

SPÉCIFICATION 2	
PROFIL	$nbSol\ :\ \mathbb{N} \to séq\,(\mathbb{N}) \to \mathbb{N}$
SÉMANTIQUE	$(nbSol\ cmpt\ s)$ est le nombre de solutions du «compte est bon» pour *cmpt* et *s*.
EX. ET PROP.	(i) $(nbSol\ 20\ [18;3;1;4;20]) = 1$ (ii) $(nbSol\ 25\ [21;3;2;4;2]) = 2$ (iii) $(nbSol\ 1\ [21;3;2;4;2]) = 0$

Q4. Étudier la fonction suivante :

SPÉCIFICATION 3	
PROFIL	$uneSol\ :\ \mathbb{N} \to séq\,(\mathbb{N}) \to séq\,(\mathbb{N})$
SÉMANTIQUE	Si $(nbSol\ cmpt\ s) \neq 0$, $(uneSol\ cmpt\ s)$ est une solution du «compte est bon» pour *cmpt* et *s*. Sinon, $(uneSol\ cmpt\ s) = [\,]$.
EX. ET PROP.	(i) $(uneSol\ 20\ [18;3;1;4;20]) = [20]$ (ii) $(uneSol\ 25\ [21;3;2;4;2]) = [21;4]$ (iii) $(uneSol\ 1\ [21;3;2;4;2]) = [\,]$

Q5. Étudier la fonction suivante :

SPÉCIFICATION 4	
PROFIL	$lesSols : \mathbb{N} \to séq(\mathbb{N}) \to séq(séq(\mathbb{N}))$
SÉMANTIQUE	$(lesSols\ cmpt\ s)$ est la séquence des solutions du « compte est bon » pour $cmpt$ et s
EX. ET PROP.	(i) $(lesSols\ 20\ [18;3;1;4;20]) = [[20]]$ (ii) $(lesSols\ 25\ [21;3;2;4;2]) = [[21;4];[21;2;2]]$ (iii) $(lesSols\ 1\ [21;3;2;4;2]) = [\,]$

Indication On pourra utiliser la fonction auxiliaire suivante :

SPÉCIFICATION 5	
PROFIL	$ajg : \alpha \to séq(séq(\alpha)) \to séq(séq(\alpha))$
SÉMANTIQUE	Soit $n \in \mathbb{N}^*$, $(ajg\ x\ [s_1;\dots;s_n]) \stackrel{def}{=} [x::s_1;\dots;x::s_n]$. Par convention, $(ajg\ x\ [\,]) = [\,]$.

7.9 TD8 Quelques tris

La spécification d'une fonction de *tri* est courte et simple :

SPÉCIFICATION 1	
PROFIL	$tri : séq(\alpha) \to séq(\alpha)$
SÉMANTIQUE	tri(s) est une séquence formée des éléments de s énumérés en ordre croissant $(\leqslant)$
EX. ET PROP.	(i) $tri\ ([3;-2;5;3;1]) = [-2;1;3;3;5]$

Il existe plusieurs dizaines de réalisations de cette spécification. De la plus efficace, smoothsort [2] (plus rapide que le tri rapide [3] dans certains cas particuliers), à la plus complexe (en termes de structures de données), le tri arborescent [4] (voir partie 4 de ce cours), en passant par celle de complexité asymptotique optimale, le tri par tas [5] . Mentionnons la plus simple, le tri stupide [6] , et la plus inattendue, gnome sort [7] (d'après la méthode qu'utilisent les nains de jardin Hollandais pour trier une rangée de pots de fleurs).

Commençons par la plus classique, le tri par insertion [8] (c'est l'algorithme adopté de manière naturelle aux cartes par les joueurs triant leur main avant de commencer le jeu).

[2] https://fr.wikipedia.org/wiki/Smoothsort
[3] https://fr.wikipedia.org/wiki/Tri_rapide
[4] https://fr.wikipedia.org/wiki/Tri_arborescent
[5] https://fr.wikipedia.org/wiki/Tri_par_tas
[6] https://fr.wikipedia.org/wiki/Tri_stupide
[7] https://en.wikipedia.org/wiki/Gnome_sort
[8] https://fr.wikipedia.org/wiki/Tri_par_insertion

7.9.1 Tri par insertion

Considérons la séquence $s = pr :: fin$. Le principe du tri par insertion consiste à trier fin, puis y insérer pr au bon endroit (selon $\leqslant$).

Q1. Donner des équations définissant *tri* selon le principe énoncé. Spécifier et réaliser d'éventuelles fonctions intermédiaires.

7.9.2 Tri par sélection

Le tri par sélection du minimum [9] est simple, mais inefficace. Il est néanmoins à connaître, du fait de sa simplicité.

Soit s une séquence, m la valeur du plus petit élément de s, et $s' = s \backslash \{m\}$. On obtient une séquence triée en plaçant m en tête de la séquence obtenue en triant s'.

Q2. Donner des équations définissant *tri* selon le principe énoncé. Spécifier et réaliser les éventuelles fonctions intermédiaires, en prenant soin d'éviter les calculs redondants : il est inutile de parcourir s deux fois pour obtenir m et s'.

7.9.3 Tri pivot («quicksort»)

Application du principe «diviser pour régner», le tri pivot [10] (ou tri rapide, ou encore tri par partition) est un tri efficace (tant qu'on ne se place pas dans le pire des cas), et donc très utilisé. C'est lui, ou une de ses variantes, qui est en général implémenté dans les librairies standards des langages de programmation.

Soit s une séquence non vide, à partir de laquelle on produit trois objets : un élément p - le *pivot* - et deux séquences *inf* et *sup* définis comme suit :

- p est le premier élément de s (il y a d'autres façons de choisir un pivot [10]),
- *inf* est une séquence formée des éléments de $s \leqslant p$,
- *sup* est une séquence formée des éléments de $s > p$.

Par exemple, pour $s = [7;5;25;3;7;15;4;20]$:

- $p = 7$,
- $inf = [5;3;7;4]$
- $sup = [25;15;20]$.

On obtient une séquence triée formée des éléments de s en concaténant la séquence triée des éléments de *inf*, le pivot et la séquence triée des éléments de *sup*.

Q3. Donner des équations définissant la fonction *tri*selon le principe énoncé. Spécifier et réaliser les fonctions intermédiaires nécessaires.

[9] https://fr.wikipedia.org/wiki/Tri_par_sélection
[10] https://fr.wikipedia.org/wiki/Tri_rapide

7.10 TD8 Anagrammes

Deux mots sont des *anagrammes* l'un de l'autre, s'ils comportent exactement les mêmes lettres, éventuellement dans un ordre différent. Par exemple, « sel » et « les » sont des anagrammes, de même que « anis » et « sain », « arbre » et « barre », « écran » et « nacré », « arsenelupin », « paulsernine » et « luisperenna ».

Par contre « lasse » et « salle » ne sont pas des anagrammes.

On étudie la fonction suivante :

SPÉCIFICATION 1	
PROFIL	$estAna \ : \ séq\,(\alpha) \to séq\,(\alpha) \to \mathbb{B}$
SÉMANTIQUE	$(estAna\ s_1\ s_2) = vrai$ si et seulement si s_1 et s_2 sont des anagrammes.

Une solution est fondée sur le principe suivant : chaque lettre de la première séquence doit appartenir à la seconde et la seconde ne doit pas en comporter d'autres.

On spécifie donc la fonction intermédiaire suivante :

SPÉCIFICATION 2	
PROFIL	$appsuppr \ : \ \alpha \to séq\,(\alpha) \to \mathbb{B} \times séq\,(\alpha)$
SÉMANTIQUE	Posons $b, s' = (appsuppr\ x\ s)$; b est vrai si et seulement si $x \in s$, et dans ce cas $s' = s \backslash \{1^{re}\ occurrence\ de\ x\}$. Si $x \notin s$, alors $b = faux$ et s' est sans objet.

Q1. Réaliser *appsuppr*.

Q2. En déduire une réalisation de *estAna*.

7.11 TD8 Concaténation

On connait l'opérateur *infixe* de concaténation de deux séquences, noté `(@)` en OCaml. Dans cet exercice, on étudie la concaténation sous la forme d'une fonction *préfixe* appelée *concat*.

Q1. Spécifier *concat*.

Q2. Compléter les équations suivantes :

(1) $(concat\ [\,]\ [\,]) = \ldots\ldots\ldots$

(2) $(concat\ [\,]\ s_2) = \ldots\ldots\ldots$

(3) $(concat\ s_1\ [\,]) = \ldots\ldots\ldots$

(4) $(concat\ pr_1 :: fin_1\ s_2) = \ldots\ldots\ldots\ldots\ldots\ldots$

Les équations 1 à 4 donnent des propriétés qui sont toutes correctes, au sens où l'évaluation de la partie gauche de l'équation donne un résultat égal à l'évaluation de la partie droite. En revanche, les différentes combinaisons de ces propriétés ne donnent pas toujours un algorithme correct, au sens expliqué dans les questions suivantes.

Q3. Expliquer, en donnant un exemple, pourquoi la combinaison des équations $\{1,3,4\}$ est insuffisante.

Q4. Expliquer en quoi la combinaison des équations $\{1,2,4\}$ n'est pas mininale.

Q5. Donner une combinaison dans laquelle chaque propriété est nécessaire et suffisante par rapport aux autres propriétés de la combinaison.

Q6. En déduire une réalisation de *concat*.

7.12 TD8 Préfixe

Une séquence, dénotée *pré*, est un préfixe d'une séquence s si et seulement si il existe une séquence, dénotée *suite*, telle que $s = pré \mathbin{@} suite$.

La séquence vide est préfixe de toute séquence ; toute séquence est préfixe d'elle-même.

Q1. Définir un prédicat nommé *estpref* portant sur deux séquences et ayant la valeur vrai lorsque la première est un préfixe de la seconde.

Q2. Définir le prédicat *egseq* qui teste l'égalité de deux séquences. Comparer les équations récursives de *estpref* et *egseq*.

Q3. Définir une fonction *lespref* qui construit une séquence comportant **tous** les préfixes d'une séquence d'entiers donnée. Exemple : *lespref* $([1;2;3]) = [[\,];[1];[1;2];[1;2;3]]$.

Troisième partie

ORDRE SUPÉRIEUR

Chapitre 8

Schémas et fonctions d'ordre supérieur

Sommaire

Dans le titre de chaque exercice, problème ou annale d'examen est indiqué TDn ou TPn , où n est un entier entre 1 et 12 correspondant au numéro de séance de TD ou TP **à partir duquel** l'énoncé peut être abordé. En pratique, un exercice n^o n sera peut-être traité en séance $n+1$ ou $n+2$, et les exercices ne sont pas forcément traités dans l'ordre du polycopié.

Il y a plus d'énoncés que de séances ; ne pas hésiter à s'entraîner sur les énoncés qui n'auront pas été abordés en séance, notamment pendant le 2^e créneau de TP (non supervisé).

8.1 TD9 Fonctions d'ordre sup. simples

Cet exercice consiste à réaliser des fonctions d'ordre supérieur simples illustrant certains concepts mathématiques de base.

Q1. Spécifier (on donnera un exemple et un contre-exemple) puis réaliser une fonction *estPtf x* qui indique si un élément donné est un point fixe d'une fonction donnée.

> **Indication** Un élément x appartenant au domaine de définition d'une fonction f est un point fixe de f si et seulement si $f(x) = x$.

Q2. Spécifier (on donnera un exemple et un contre-exemple) puis réaliser une fonction *estIdempot* qui indique si une fonction donnée est idempotente sur un domaine de valeurs donné (fourni par la liste des éléments du domaine).

> **Indication** Une fonction f est idempotente sur un domaine D si et seulement si $\forall x \in D$, $f(f(x)) = f(x)$.

8.2 TD9 Composition, fonction locale

En mathématiques, la composition de fonctions est notée grâce à l'opérateur infixe $\circ$. Pour toutes fonctions f et g telles que g soit définie sur l'ensemble des valeurs prises par f, $(g \circ f)(x) \stackrel{def}{=} g(f(x))$.

En informatique, et surtout en programmation fonctionnelle, on peut être amené à composer beaucoup de fonctions (des dizaines dans certains contextes applicatifs). Ainsi, l'usage[1] veut qu'on précise d'abord `f`, puis `g`, contrairement à la notation mathématique ; si `comp` est une fonction implémentant la composition, `(comp f g)` et $g \circ f$ sont alors deux notations différentes pour le même objet.

Q1. Spécifier et implémenter *comp*.

OCaml permet à l'utilisateur de définir ses propres opérateurs binaires infixes. La syntaxe est la même que celle pour définir une fonction préfixe, à la différence que :

- l'identificateur de l'opérateur est une suite de symboles parmi `! $ % & * + - . / : ? @ ^ < > = | ~` (donc pas de lettre ni de chiffre, mais plusieurs symboles peuvent être accolés) ;
- l'opérateur doit être placé entre parenthèses, par exemple :
  - définition : `let (@|) (x:…) (y:…) = …`
  - uilisation : … `x @| y` …

Q2. Définir un opérateur infixe $\gg$ implémentant la composition.

Q3. Définir, directement puis par application partielle, une fonction *suiv* qui, à un entier donné, associe le suivant. En déduire l'implémentation d'une fonction `plus2` qui ajoute 2 à un entier, en utilisant uniquement $\gg$ et *suiv*.

Q4. Définir, grâce à une application partielle, une fonction *fois*2 qui multiplie un entier donné par 2.

Q5. En utilisant les fonctions *suiv* et *fois*2, définir :

- la fonction $f1$ qui à chaque entier x associe $2 * x + 1$,
- la fonction $f2$ qui à chaque entier x associe l'entier $2 * (x + 1)$

Pour chacune de ces deux fonctions, proposer une variante utilisant $\gg$ et l'autre ne l'utilisant pas.

8.3 TD9 Définition des schémas *map* et *fold*

Nous allons définir deux *schémas* d'ordre supérieur très utilisés en programmation fonctionnelle, à travers les fonctions *map* et *fold* (aussi appelé *reduce* dans d'autres langages).

Ces fonctions, ainsi que beaucoup d'autres schémas d'ordre supérieur, sont prédéfinies par OCaml dans le module `List`.

Les éléments d'une séquence non vide s de longueur n seront notés $e_0, \ldots, e_{n-1}$. Autrement dit : $s = [e_0 \,;\, \ldots \,;\, e_{n-1}]$. Afin de simplifier les notations, nous conviendrons que si $n = 0$, s est la séquence

[1] largement influencé par une culture occidentale lisant et écrivant de gauche à droite

vide [ ].

Q1. Rappeler la spécification puis réaliser la fonction *map* vue en cours.

Q2. Rappeler la spécification puis réaliser la fonction $fold\_left$ vue en cours.

La fonction $fold\_left$ consomme les éléments de la séquence sur laquelle elle est appliquée de e_0 à e_{n-1}, c'est-à-dire par la gauche (d'où son nom).

Q3. Spécifier puis réaliser une fonction $fold\_right$ consommant les éléments de la séquence sur laquelle elle est appliquée par la droite, c'est-à-dire de e_{n-1} à e_0.

8.4 TD9 Applications de *map* et $fold$

On rappelle les schémas d'ordre supérieur *map* et $fold$:

SPÉCIFICATION 1	
PROFIL	$map\ :\ (\alpha \to \beta) \to séq\,(\alpha) \to séq\,(\beta)$
SÉMANTIQUE	$(map\ f\ [e_0\ ;\ \ldots\ ;\ e_{n-1}]) = [f\,e_0\ ;\ \ldots\ ;\ f\,e_{n-1}]$

SPÉCIFICATION 2	
Étant donné une fonction de combinaison *co* et une valeur initiale *init* :	
PROFIL	$fold\_left\ :\ (\beta \to \alpha \to \beta) \to \beta \to séq\,(\alpha) \to \beta$
SÉMANTIQUE	$(fold\_left\ co\ init\ [e_0\ ;\ \ldots\ ;\ e_{n-1}]) = (co\ \cdots\ (co\ (co\ init\ e_0)\ e_1)\ \cdots\ e_{n-1})$
PROFIL	$fold\_right\ :\ (\alpha \to \beta \to \beta) \to séq\,(\alpha) \to \beta \to \beta$
SÉMANTIQUE	$(fold\_right\ co\ [e_0\ ;\ \ldots\ ;\ e_{n-1}]\ init) = (co\ e_0\ (co\ e_1\ (\cdots\ (co\ e_{n-1}\ init)\cdots)))$

Dans cet exercice, aucune des fonctions dont la réalisation est demandée n'est récursive ; on devra utiliser un (ou plusieurs) des schémas ci-dessus pour les implémenter.

8.4.1 Quelques manipulations arithmétiques élémentaires

Q1. Définir une fonction qui incrémente chaque entier d'une séquence. On proposera une implémentation « normale » utilisant une fonction anonyme, une implémentation par application partielle d'un schéma, ainsi qu'une implémentation sans fonction anonyme.

Q2. Définir une fonction qui construit la séquence des résultats de la division euclidienne par un entier non nul donné (quotient et reste), de chaque entier d'une séquence.

8.4.2 Conversion d'une séquence de caractères en chaîne

Q3. Implémenter une fonction *cVs* qui convertit un caractère en chaîne par application partielle de la fonction `make` du module `String`, de profil `int -> char -> string` : `(make n c)` construit une chaîne contenant `n` fois le caractère `c`.

Q4. Définir une fonction qui convertit une séquence de caractères en chaîne.

Q5. Quel est le résultat de l'évaluation de l'expression : `fold_right (fun e acc -> acc ^ (cVs e)) ['o';'c';'a';'m';'l'] ""` ?

8.4.3 Maximum d'une séquence

Q6. Définir une fonction *maxs* qui produit le maximum d'une séquence de **naturels** ($\mathbb{N}$) donnée, à l'aide :

a) d'une fonction anonyme,

b) de la fonction OCaml `max`, de profil $\alpha \to \alpha \to \alpha$, qui donne le plus grand de ses arguments.

Q7. Comment faire si les entiers de la séquence ne sont pas bornés inférieurement ?

8.4.4 Inversion d'une séquence

Q8. Définir une fonction permettant d'inverser une séquence. On donnera deux implémentations utilisant des schémas distincts.
NB : cette fonction est prédéfinie par OCaml dans le module `List` sous le nom `rev`.

8.4.5 I^e élément d'une séquence

Le i^e élément d'une séquence est spécifié ainsi :

SPÉCIFICATION 3	
PROFIL	$ieme$: $\mathbb{N}^* \to séq\,(\alpha)^* \to \alpha$
SÉMANTIQUE	$(ieme\ i\ s)$ est le i^e élément de s. Précondition : $i \leqslant \lvert s \rvert$
EX. ET PROP.	(i) $(ieme\ 1\ [2;1;4]) = 2$ (ii) $(ieme\ 2\ [2;1;4]) = 1$ (iii) $(ieme\ 3\ [2;1;4]) = 4$

Q9. Donner une implémentation de la fonction *ieme* en utilisant le schéma d'odre supérieur $fold_{right}$ et les sélecteurs `List.hd` et `List.tl`.

> **Indication** Soit s une séquence dont on veut le i^e élément ; l'accumulateur de la fonction de combinaison est un couple formé de l'index courant dans s (initialement 1) et d'une séquence (initialement s), à déterminer.

8.4.6 Applatissement d'une séquence de séquences

L'aplatissement d'une séquence de séquences est spécifié ainsi :

SPÉCIFICATION 4	
PROFIL	$aplat$: $séq\,(séq\,(\alpha)) \to séq\,(\alpha)$
Notons que α étant un type quelconque, il peut être lui aussi une séquence ...	
SÉMANTIQUE	$(aplat\ [s_0\,;\cdots;\,s_{n-1}]) = s_0 @ \cdots @ s_{n-1}$
EX. ET PROP.	(i) $aplat([\,[1;2;3]\,;\,[4]\,;\,[5;6;7]\,;\,[\,]\,;\,[8;9]\,;\,[\,]\,]) = [1;2;3;4;5;6;7;8;9]$ (ii) $aplat([\,[[1;2];[3;4]]\,;\,[[5]]\,;\,[[6;7];[8;9;10]]\,] =$ $[[1;2];[3;4];[5];[6;7];[8;9;10]])$

Q10. Donner une Implémentation non récursive de *aplat*.
NB : cette fonction est prédéfinie par OCaml dans le module `List` sous le nom `flatten`.

8.4.7 Implémentation de *map* par *fold*

Q11. Implémenter la fonction *map* grâce à un des schémas *fold*.

8.5 TD10 Soyons logiques

8.5.1 ∀ et ∧

Q1. Définir la fonction *conj* (le « et », noté ∧ en math) : appliquée à une séquence de booléens, *conj* retourne vrai si et seulement si tous les éléments de la séquence sont vrais.
L'implémentation devra être non récursive.

Étant donnés une séquence s et un prédicat p, la fonction *for_all*, prédéfinie par OCaml dans le module `List`, implémente le quantificateur universel (∀). Ainsi, $(for\_all\ p\ s) = vrai$ si et seulement si tous les éléments de s vérifient p.

Q2. En déduire une autre implémentation (non récursive) de *conj*.

On souhaite maintenant faire le contraire :

Q3. Implémenter une fonction *pour_tout* en utilisant *conj* et un schéma d'ordre supérieur bien connu.

Q4. Ré-implémenter *pour_tout* grâce à un schéma *fold*.

8.5.2 ∃ et ∨

Q5. Reprendre les questions précédentes en remplaçant :

- *conj* par *disj* (disjonction, le « ou », noté ∨ en math),
- *pour_tout* par *existe* (∃).

8.5.3 ¬

Q6. Définir la négation (le « non », noté ¬ en math).

Q7. Que vous inspire la propriété logique : $\neg\forall x\ A(x) \equiv \exists x\ \neg A(x)$?

8.6 TD10 Définition des schémas *find*, *filter* et *partition*

On rappelle l'implémentation de $option(\alpha)$:

```
type 'a option = Some of 'a | None
```

Q1. Spécifier et réaliser un schéma d'ordre supérieur appelé *find_opt* qui retourne le premier élément d'une séquence s donnée satisfaisant un prédicat p donné, ou `None` si aucun élément de s ne satisfait p.
NB : ce schéma est prédéfini dans le module `List`.

Q2. En déduire une implémentation non récursive de la fonction d'appartenance spécifiée ci-dessous.

SPÉCIFICATION 2	
PROFIL	`app` : $\alpha \rightarrow \mathit{séq}\,(\alpha) \rightarrow \mathbb{B}$
SÉMANTIQUE	$(\mathit{app}\ x\ s)$ est vrai si et seulement si $x \in s$.

NB : cette fonction est prédéfinie dans le module `List` sous le nom `mem`.

Q3. Spécifier et réaliser un schéma d'ordre supérieur *filter* qui retourne les éléments d'une séquence donnée satisfaisant un prédicat donné. L'ordre des éléments de la liste doit être conservé.
NB : ce schéma est prédéfini dans le module `List`.

Q4. En déduire une autre implémentation (non récursive) de `app`.

Étant donné un prédicat p et une séquence s, on souhaite produire deux séquences s_1 et s_2 définies comme suit :

- s_1 liste les éléments de s qui satisfont p,
- s_2 liste les éléments de s qui ne satisfont pas p.

L'ordre des éléments de s doit être conservé.

Q5. Spécifier un schéma d'ordre supérieur appelé *partition* qui effectue l'opération décrite ci-dessus.
NB : ce schéma est prédéfini dans le module `List`.

Q6. Réaliser *partition* (équations récursives, terminaison, implémentation).

Q7. Donner une implémentation non récursive de *partition*.

8.7 TD10 Tri rapide par pivot (quicksort)

Étant donnée une séquence s, le principe du tri par pivot consiste à :

1. choisir un élément de s qu'on appelera le *pivot*,
2. filtrer la séquence s en deux sous-séquences s_1 et s_2 où :
   - s_1 est composée des éléments de s inférieurs ou égaux au pivot,
   - s_2 est composée des éléments de s strictement supérieurs au pivot,
3. trier s_1 et s_2,
4. construire la séquence résultat (triée) en plaçant le pivot entre les deux séquences (déjà triées) obtenues au 3.

Dans la suite, on choisira de prendre pour pivot le premier élément de la séquence.

Q1. Soit $s = [3;5;1;0;4;2]$. Que valent le pivot, s_1 et s_2 ?

Q2. Rappeler la spécification de deux schémas d'ordre supérieur filtrant les éléments d'une séquence.

Q3. En déduire deux implémentations d'une fonction de tri basée sur l'algorithme du pivot.

8.8 TD10 Somme de pieces des monnaies

On souhaite énumérer tous les montants que l'on peut réaliser avec un exemplaire de chaque pièce de la monnaie européenne.

Ce problème se pose lors de la conception d'une nouvelle monnaie afin de déterminer quelles sont les pièces essentielles pour obtenir tous les montants (centime par centime dans le cas de l'euro, de 0 à 5€) avec un minimum de pièces.

On définit l'ensemble des montant d'une pièce : $pièce \stackrel{def}{=} \{0.01,\ 0.02,\ 0.05,\ 0.10,\ 0.20,\ 0.50,\ 1,\ 2\}$

Q1. Implémenter le type *piece* et la séquence constante `les_pieces` des pièces de la monnaie européenne.

On spécifie la fonction auxiliaire suivante :

SPÉCIFICATION 1	
PROFIL	$ajoutAchq\ :\ \alpha \to séq\ (séq\ (\alpha)) \to séq\ (séq\ (\alpha))$
SÉMANTIQUE	$(ajoutAchq\ x\ ss)$ est la séquence obtenue en ajoutant x en queue de chaque séquence de ss.
EX. ET PROP.	(i) $(ajoutAchq\ 0\ [[2;3];[5];[\,];[3;8]])\ =\ [\ [2;3;0]\ ;\ [5;0]\ ;\ [0]\ ;\ [3;8;0]\ ]$

Q2. Donner une implémentation non récursive d'*ajoutAchq*.

L'ensemble $\mathcal{P}(E)$ des parties d'un ensemble E est spécifié de la manière suivante :

SPÉCIFICATION 2	
PROFIL	$part\ :\ séq\ (\alpha) \to séq\ (séq\ (\alpha))$
SÉMANTIQUE	$part(s)$ est l'ensemble des parties de s.
EX. ET PROP.	(i) $part([\,]) = [[\,]]$ (ii) $part([e_0]) = [[\,]\ ;\ [e_0]]$ (iii) $part([e_0;e_1;e_2]) = [\ [\,]\ ;\ [e_0]\ ;\ [e_1]\ ;\ [e_2];\ [e_0;e1]\ ;\ [e_0;e_2]\ ;\ [e_1;e_2]\ ;\ [e_0;e_1;e_2]\ ]$

Q3. Donner une implémentation non récursive de *part*.

Q4. Donner une implémentation non récursive de la fonction :

SPÉCIFICATION 3	
PROFIL	somme : *séq* (*piece*) $->$ $\mathbb{R}^+$
SÉMANTIQUE	(*somme s*) est la somme des pièces de *s*.

Afin d'éviter les aléas classiques des calculs en virgule flottante inhérents à la norme IEEE 754 [2] (par exemple : `0.02+.0.2+.0.5+.1.+.2. = 3.71999999999999975`), on calculera les sommes partielles sur des entiers.

Q5. Déduire des questions précédentes une implémentation non récursive de la fonction :

SPÉCIFICATION 4	
PROFIL	*combinaisons* : *séq* (*piece*) $\to \mathbb{R} \times$ *séq* (*séq* (*piece*))
SÉMANTIQUE	Soit *s* un porte-monnaie ; *combinaisons*(*s*) est l'ensemble de tous les couples de la forme (m, p) où *p* est une partie de *s* (les pièces utilisées) et *m* le montant obtenu avec ces pièces.
EX. ET PROP.	(i) *combinaisons*([0.1; 0.2]) = [0.0, [ ] ; 0.1, [0.1] ; 0.2, [0.2] ; 0.3, [0.1; 0.2]]

[2] https://fr.wikipedia.org/wiki/IEEE_754

Quatrième partie

STRUCTURES ARBORESCENTES

Chapitre 9

Rappels

Vocabulaire

- *Définir* = donner une définition ; *définition* = spécification + réalisation
- *Spécifier* = donner une spécification (le « quoi ») ; *pécification* = profil + sémantique + examples et/ou propriétés
- *Réaliser* = donner une réalisation (le « comment ») ; *réalisation* = algorithme (langue naturelle et/ou équations récursives + terminaison) + implémentation (OCaml)
- *Implémenter* = donner une inplémentation (OCaml)

Dans certains cas, certaines de ces rubriques peuvent être omises.

9.1 Principe de définition d'une fonction récursive

Ce principe est général. Déjà présenté dans la partie « Définition récursives », il s'applique également aux structures arborescentes.

> « On s'appuie sur la **structure récursive** de l'ensemble de départ de la fonction. »

Chapitre 10

Fonctions récursives sur les arbres binaires

Sommaire

Dans le titre de chaque exercice, problème ou annale d'examen est indiqué TDn ou TPn , où n est un entier entre 1 et 12 correspondant au numéro de séance de TD ou TP **à partir duquel** l'énoncé peut être abordé. En pratique, un exercice n^o n sera peut-être traité en séance $n+1$ ou $n+2$, et les exercices ne sont pas forcément traités dans l'ordre du polycopié.

Il y a plus d'énoncés que de séances ; ne pas hésiter à s'entraîner sur les énoncés qui n'auront pas été abordés en séance, notamment pendant le 2^e créneau de TP (non supervisé).

10.1 TD11 Des ensembles aux arbres

On définit la *profondeur* d'un arbre comme le nombre d'éléments que contient sa branche la plus longue.

On considère un ensemble S de 5 éléments quelconques $\{e_1, e_2, e_3, e_4, e_5\}$.

Q1. Représenter graphiquement plusieurs arbres binaires dont l'ensemble des nœuds est l'ensemble S :

a) arbre de profondeur minimale,

b) arbre de profondeur maximale.

Donner pour chacun d'eux diverses notations OCaml.

Q2. Généraliser : quelle est la profondeur maximale d'un arbre contenant n nœuds ?

10.2 TD11 Nombre de nœuds et profondeur

Q1. Représenter graphiquement des arbres de profondeur 0, 1, 2 et 3 de deux manières :

a) en dessinant l'arbre vide (graphiquement représenté par $\triangle$),

b) sans dessiner l'arbre vide.

Pour chaque arbre, on donnera l'implémentation en OCaml.

Chaque nœud sera représenté par $\bullet$. On note $p(a)$ la profondeur d'un arbre a.

	$p(a)=0$	$p(a)=1$	$p(a)=2$	$p(a)=3$
a) avec l'arbre vide	$\triangle$	$\vdots$	$\vdots$	$\vdots$
	`Av`	...	...	...
b) sans l'arbre vide		$\vdots$	$\vdots$	$\vdots$
		...	...	...

Q2. Compléter le tableau ci-dessous. On note $n(a)$ le nombre de nœuds d'un arbre a.

profondeur	nombre de nœuds
$p(a)=0$	$.. \leqslant n(a) \leqslant ..$
$p(a)=1$	$.. \leqslant n(a) \leqslant ..$
$p(a)=2$	$.. \leqslant n(a) \leqslant ..$
$p(a)=3$	$.. \leqslant n(a) \leqslant ..$

Q3. En déduire les nombres minimal et maximal de nœuds que peut contenir un arbre en fonction de sa profondeur p.

10.3 TD11 Profondeur d'un arbre

Q1. Donner la spécification + réalisation complète (équations de récurrence, terminaison, implémentation) d'une fonction polymorphe renvoyant la profondeur d'un arbre.

10.4 TD11 Appartenance

Q1. Spécifier un prédicat polymorphe *appArb* d'appartenance d'un élément à un arbre.

Q2. Donner des équations récursives définissant *appArb*, puis justifier que $\forall e \in \alpha, \forall a \in abin(\alpha)$, l'évaluation de $(appArb\ e\ a)$ termine

Q3. Implémenter *appArb* sans utiliser d'expression conditionnelle.

Q4. Implémenter *appArb* sans utiliser d'expression conditionnelle ni de filtrage : seule la composition booléenne est autorisée.

Indication On pourra utiliser les sélecteur suivants :

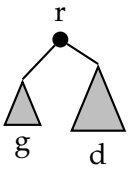
SPÉCIFICATION 1	
PROFIL	rac : $abin(\alpha)\backslash\{\triangle\} \to \alpha$
SÉMANTIQUE	$racine(a)$ est la racine de a.
PROFIL	gauche : $abin(\alpha)\backslash\{\triangle\} \to abin(\alpha)$
SÉMANTIQUE	$gauche(a)$ est le sous-arbre gauche de a.
PROFIL	droit : $abin(\alpha)\backslash\{\triangle\} \to abin(\alpha)$
SÉMANTIQUE	$droit(a)$ est le sous-arbre droit de a.

Q5. Définir (spécification + réalisation) une fonction qui aplatit un arbre, c'est-à-dire le transforme en une séquence de ses nœuds.

Q6. En déduire une autre réalisation de *appArb*, non récursive.

10.5 TD12 Niveau d'un nœud

Le niveau d'un nœud e dans un arbre est le nombre de nœuds sur la branche qui conduit de la racine de l'arbre jusqu'à e inclus. La racine est donc de niveau 1.

Soit $p \in \mathbb{N}^*$. On considère un nœud e de niveau $p+1$ dans un arbre 

.

Alors e se situe soit dans g, soit dans d, et il est de niveau p dans le sous-arbre auquel il appartient (g ou d).

10.5.1 Nombre de feuilles à un niveau donné

Q1. Spécifier une fonction *nbf_deniv* qui donne le nombre de feuilles à un niveau donné dans un arbre.

Q2. Donner des équations récursives définissant *nbf_deniv*, et étudier le problème de la terminaison.

Q3. Implémenter *nbf_deniv*.

10.5.2 Niveau d'un élément

Q4. Donner une spécification + réalisation complète (équations de récurrence, terminaison, implémentation) de la fonction *niv* qui donne le niveau d'un élément dans un arbre ; on conviendra que le niveau d'un élément qui n'est pas présent dans l'arbre est 0.