

**UNIVERSITÉ
GRENOBLE
ALPES**

UFR IM²AG

**DÉPARTEMENT
LICENCE SCIENCES ET
TECHNOLOGIE**

**LICENCE SCIENCES & TECHNOLOGIES
1^{re} ANNÉE**

**UE INF201, INF231
ALGORITHMIQUE ET PROGRAMMATION FONCTIONNELLE**

2019 / 2020

EXERCICES DE TRAVAUX DIRIGÉS

Dans le titre de chaque exercice, problème ou annale d'examen est indiqué **TD n** ou **TP n** , où n est un entier entre 1 et 12 correspondant au numéro de séance de travaux dirigés ou pratiques **à partir duquel** l'énoncé peut être abordé.

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.

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

Table des matières

Première partie	TYPES, EXPRESSIONS ET FONCTIONS	2
1	Appropriation des notations	3
1.1	TD1 Notation des types et des valeurs	3
1.2	TD1 Opérations logiques	6
1.3	TD1 Opérations arithmétiques et logiques	7
1.4	TD1 Chaînes de caractères	8
1.5	TD1 Expressions conditionnelles	9
1.6	TD1 Vérification des types dans une expression	10
2	Types, expressions et fonctions	12
2.1	TD1 Signe du produit	12
2.2	TD1 Une date est-elle correcte ?	14
2.3	TD1 Quelle heure est-il ?	15
2.4	TD2 Fractions	16
2.5	TD2 Géométrie élémentaire	17
2.6	TD2 Relations sur des intervalles d'entiers	18
2.6.1	Relations entre points et intervalles	18
2.6.2	Relations entre intervalles	19
2.7	TD3 Le code de César	20
2.7.1	Version 1	20
2.7.2	Version 2	20
2.7.3	Généralisation	21
2.8	TD3 Nomenclature de pièces	21
2.8.1	Types associés à la notion de référence	22
2.8.2	Fonctions associées à la notion de référence	22
2.8.3	Comparaison de références : relation d'ordre sur les types	23
3	Annales	25
3.1	Partiel 14–15 TD3 « un dîner presque parfait »	25
3.1.1	Modélisation	25
3.1.2	Coûts des éléments du repas	26
3.1.3	À table	26
3.2	Partiel 13–14 TD3 colorimétrie	27
3.2.1	Colors	28
3.2.2	The grey color	28

3.2.3	Barbouillons	28
3.2.4	Grey and other colors	29
Deuxième partie DÉFINITIONS RÉCURSIVES		31
4	Types récursifs	32
4.1	TD4 Puissances d'entiers	32
4.1.1	Définition des types	33
4.1.2	Racine et l'exposant d'une valeur de type <i>puisrec</i>	33
4.1.3	Conversions d'un type vers un autre	34
4.1.4	Produit de deux puissances	35
4.2	TD5 Séq. construites par la droite	36
4.2.1	Type récursif « séquence d'entiers »	36
4.3	TD5 Séq. construites par la gauche	36
4.3.1	Type « séquence d'entiers »	36
4.3.2	Dernier entier	36
4.3.3	Premier élément	37
4.3.4	Séquence de caractères	37
4.3.5	Séquences de couples d'entiers	37
4.3.6	Généralisation : liste de type quelconque	38
4.4	TD6 Le type séquence prédéfini d'OCAML	38
4.5	TD6 Séquences	39
4.6	TD6 Séq. de séq. ou de nuplets	40
5	Fonctions récursives sur les séquences	41
5.1	TD6 Un élément apparaît-il dans une séq.	42
5.2	TD6 Nombre d'occ. d'un élément	43
5.3	TD6 Maximum de n entiers	43
5.4	TD6 Suppression des espaces	44
5.5	TD6 Plus de 'a' que de 'e' ?	45
5.6	TD7 Jouons aux cartes	46
5.6.1	Valeur d'un joker	46
5.6.2	Valeur d'une petite	46
5.6.3	Valeur d'un honneur	46
5.6.4	Valeur d'une main	47
5.7	TD7 Ordre sur des mots ?	47
5.8	TD7 Séq. des valeurs cumulées	47
5.8.1	Réalisation avec découpage et construction de séquence par la droite	47
5.8.2	Découpage à droite de listes construites par la gauche **	48
5.8.3	Découpage à gauche et listes construites par la gauche *	49
5.8.4	Réalisation directe sans fonction intermédiaire *	50
5.9	TD7 *** Le compte est bon	50
5.10	TD8 Quelques tris	51
5.10.1	Tri par insertion	51
5.10.2	Tri par sélection du minimum	51

5.10.3	Tri par pivot/partition (« quicksort »)	51
5.11	TD8 Anagrammes	52
5.11.1	Première réalisation : deux fonctions intermédiaires	52
5.11.2	Seconde réalisation : regroupement des fonctions intermédiaires	52
5.12	TD8 Concaténation de deux séq.	52
5.13	TD8 Préfixe d'une séquence	53
6	Annales	54
6.1	Partiel 14–15 TD8 « gardez la monnaie »	54
6.1.1	Modélisation d'un porte-monnaie	54
6.1.2	Somme d'argent et porte-monnaie	55
6.1.3	Shopping	56
6.1.4	Porte-monnaie et séquences de pièces	56
6.2	Exam 14–15 TD8 décomposition en facteurs premiers	57
6.3	Exam 14–15 TD8 relations binaires	59
	Troisième partie ORDRE SUPÉRIEUR	62
7	Ordre supérieur	63
7.1	TD9 Fonctions d'ordre sup. simples	63
7.2	TD9 Composition, fonction locale	64
7.3	TD9 Utilisation des fonctions d'ordre sup. <i>map</i> et <i>fold</i>	64
7.4	TD9 Autre application des fonctions <i>map</i> et <i>fold</i>	65
7.4.1	Soyons logiques	65
7.4.2	Affixes	65
7.5	TD9 Tri rapide par pivot	66
7.6	TD10 Définition des fonctions d'ordre sup. <i>map</i> et <i>fold</i>	66
7.7	TD10 Somme de pièces des monnaies	68
8	Annales	69
8.1	Exam 11–12 : exercise, the omega function	69
8.1.1	Définition de <i>omega</i>	70
8.1.2	Échantillonnage d'une fonction	70
8.1.3	Extension d' <i>omega</i> aux séquences	70
8.2	Exam 13–14 : exercise, the omega function, again !	71
8.2.1	Definition of Ω	71
8.2.2	Using the Ω function	72
8.2.3	Products of lists	72
8.3	Exam 13–14 : exercise, the filter function	73
	Quatrième partie STRUCTURES ARBORESCENTES	75
9	Rappels de cours	76
9.1	Principe de définition d'une fonction récursive	76

10 Fonctions réc. sur les arbres bin.	77
10.1 TD11 Des ensembles aux arbres	77
10.2 TD11 Nombre de nœuds et prof.	78
10.3 TD11 Profondeur d'un arbre	78
10.4 TD11 Présence d'un élément	78
10.5 TD12 Descendance	79
10.5.1 Réalisation de <i>estDesc</i> à l'aide de fonctions intermédiaires	79
10.5.2 Réalisation directe de <i>estDesc</i>	80
10.6 TD12 Propriétés des arbres	80
10.7 TD12 Niveau d'un nœud	81
10.8 TD12 Ascendants d'un nœud	81
10.9 TD12 Propriétés des nœuds d'un arbre	81
10.10 TD12 Dico sous forme arborescente	84
11 Annales	87
11.1 Ex. exam 14–15 : Decision tree	87
11.2 Ex. exam 13–14 : all paths are leading to Rome	89
11.2.1 Terminology and definitions	89
11.2.2 The goal of the exercise	90
11.2.3 Tree of number of occurrences	91
11.3 Ex. exam 13–14 : Binary Search Trees	92
11.4 Pb exam 11–12 : <i>n</i> -ary trees	94
11.4.1 Un type pour les arbres <i>n</i> -aires	94
11.4.2 Nombre de feuilles	95
11.4.3 Appartenance	95

Rappels

- *définir* = donner une définition,
définition = spécification + réalisation ;
- *spécifier* = donner une spécification (le « quoi »),
spécification = profil + sémantique + exemples et/ou propriétés ;
- *réaliser* = donner une réalisation (le « comment »),
réalisation = algorithme (langue naturelle) + implémentation (OCAML) ;
- *implémenter* = donner une implémentation (OCAML).

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

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

1.1	TD1 Notation des types et des valeurs	3
1.2	TD1 Opérations logiques	6
1.3	TD1 Opérations arithmétiques et logiques	7
1.4	TD1 Chaînes de caractères	8
1.5	TD1 Expressions conditionnelles	9
1.6	TD1 Vérification des types dans une expression	10

1.1 **TD1** Notation des types et des valeurs

Q1. Compléter le tableau ci-dessous :

N°	CON- -TEXTE	EXPRESSION		TYPE		VALEUR	
		littérale	OCAML	ensemble	OCAML	littérale	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 ≠ 2	↔ 3 <> 2	.	↔	↔
6	∅	2 + vrai	↔		↔	↔
7	∅	'3'	↔	↔	↔
8	∅	'a'	↔	↔	↔
9	∅	'''	↔	↔	↔
10	∅	'a' ≤ 'b' ≤ 'A'	↔	↔	↔
11	∅	x ou faux	↔	↔	↔ .
12	Soit pi = 3.14	pi + 1,0	↔	↔	↔
13	Soit k = 3	2k	↔ 2 * k	2 \mathbb{Z}	↔	↔ .

1.2 **TD1** Opérations logiques

Table de vérité des opérateurs logiques

Q1. Nous débutons avec l'égalité, notée = en OCAML comme en mathématiques. Compléter le tableau ci-dessous :

	CONTEXTE	EXPRESSION	VALEUR
N°		littérale ↔ OCAML	OCAML
1	let $a = true$	a ↔
2	let $a = false$	a ↔
3	let $a = false$	$a = vrai$ ↔
4	let $a = true$	$a = vrai$ ↔
5	let $a = false$	$\neg a$ ↔
6	let $a = true$	$\neg a$ ↔
7	let $a = false$	$a = faux$ ↔
8	let $a = true$	$a = faux$ ↔

Q2. Regardons la table de la conjonction, usuellement appelée «et». Compléter le tableau ci-dessous :

Evaluation de l'expression $a \wedge b$ (syntaxe OCAML : $a \ \&\& \ b$)

	CONTEXTE	EXPRESSION	VALEUR
N°		littérale ↔ OCAML	OCAML
9	let $a = false \ and \ b = false$	$faux \wedge faux$ ↔
10	let $a = false \ and \ b = true$	$faux \wedge vrai$ ↔
11	let $a = true \ and \ b = false$	$vrai \wedge faux$ ↔
12	let $a = true \ and \ b = true$	$vrai \wedge vrai$ ↔

Q3. Regardons la table de la disjonction, usuellement appelée «ou». Compléter le tableau ci-dessous :

Evaluation de l'expression $a \vee b$ (syntaxe OCAML : $a \ || \ b$)

	CONTEXTE	EXPRESSION	VALEUR
N°	OCAML	littérale ↔ OCAML	OCAML
13	let $a = false \ and \ b = false$	$faux \vee faux$ ↔
14	let $a = false \ and \ b = true$	$faux \vee vrai$ ↔
15	let $a = true \ and \ b = false$	$vrai \vee faux$ ↔
16	let $a = true \ and \ b = true$	$vrai \vee vrai$ ↔

Identités remarquables

On suppose que le contexte est quelconque.

Q4. En observant les tables de vérité de :

- $\neg a$,
- $a = \text{vrai}$,
- $a = \text{faux}$,

déduire une simplification de chacune des expressions suivantes :

expression littérale.	forme simplifiée
$a = \text{vrai}$.
$a = \text{faux}$...

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

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

1.3 TD1 Opérations arithmétiques et logiques

Si l'expression est typable dans le contexte proposé alors précisez son type et donnez 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

N°	CONTEXTE (math.)	EXPRESSION (OCAML)	TYPE (math. ↔ OCAML)	VALEUR (OCAML)
1	a ↦ «hibou» b ↦ 'x'	a ^ b ↔
2	a ↦ «200» b ↦ 7	a ^ "b" ↔
3	a ↦ «hib» b ↦ «0»	a ^ b ↔
4	a ↦ «_» b ↦ «_»	a ^ b ↔
5	a ↦ «hib» b ↦ «0»	a @ b ↔
6	a ↦ 'a' b ↦ «bc»	a b ↔
7	a ↦ «a» b ↦ «bc»	a b ↔
8	a ↦ 'a' b ↦ «b»	" a b" ↔
9	a ↦ «B» b ↦ «A»	a ^ b ^ b ^ a ↔
10	a ↦ " _ " b ↦ «AB»	a ^ b ^ b ^ a ↔
11	a ↦ «a» b ↦ « ^ »	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 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 *valeur absolue*

Profil $abs : \mathbb{Z} \rightarrow \mathbb{N}$

Sémantique : $abs(e)$ est la valeur absolue de e .

Exemple : $abs(-3) = 3$

RÉALISATION

Algorithme : *expression conditionnelle déterminant le signe de e*

Implémentation

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

Q2. Pourquoi cette implantation est-elle incorrecte ?

Q3. Donner une implantation correcte.

Expressions conditionnelles à valeur booléenne

Q4. On suppose que le contexte est quelconque. Donner une expression Ocaml équivalente de chacune des expressions suivantes, en utilisant uniquement les opérateurs logiques :

N°	EXPRESSION	EXPRESSION ÉQUIVALENTE OCAML
1	<code>if a then true else false</code>	<code>.</code>
2	<code>if a then false else true</code>	<code>.....</code>
3	<code>if a then b else false</code>	<code>.....</code>
4	<code>if a then true else b</code>	<code>.....</code>
5	<code>if not a then true else b</code>	<code>.....</code>

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	$a = (b \leq c)$
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

2.1	TD1	Signe du produit	12
2.2	TD1	Une date est-elle correcte ?	14
2.3	TD1	Quelle heure est-il ?	15
2.4	TD2	Fractions	16
2.5	TD2	Géométrie élémentaire	17
2.6	TD2	Relations sur des intervalles d'entiers	18
2.6.1		Relations entre points et intervalles	18
2.6.2		Relations entre intervalles	19
2.7	TD3	Le code de César	20
2.7.1		Version 1	20
2.7.2		Version 2	20
2.7.3		Généralisation	21
2.8	TD3	Nomenclature de pièces	21
2.8.1		Types associés à la notion de référence	22
2.8.2		Fonctions associées à la notion de référence	22
2.8.3		Comparaison de références : relation d'ordre sur les types	23

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 s'il est strictement négatif.

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

SPÉCIFICATION *signe du produit*

Profil `prodPos` :

Sémantique : `prodPos x y` est vrai si et seulement si le signe du produit `xy` est positif ou nul

Exemples :

1. `(prodPos -2 -3) =`
2. $\forall x \in \mathbb{Z}, (prodPos\ x\ 0) =$
3. = faux

Q2. Compléter chacune des implantations 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 *signe du produit*

Algorithme 1 : analyse par cas dirigée par le résultat de la fonction et composition conditionnelle

Implémentation 1

```
let prodPos-1 ..... =
  if ..... then
    true
  else .....
    false
```

Algorithme 2 : analyse par cas dirigée par les données de la fonction et composition conditionnelle

Implémentation 2

```
1 let prodPos-2 ..... =
2   if x > 0 then
3     .....
4     .....
5     .....
6     .....
7   else (* x ≤ 0 *)
8     .....
9     .....
```

10
 11
 12
 13
 14

Q3. L'étudiant JeSuisPlusMalin propose une implantation *prodPos_2'* identique à *prodPos_2* mais sans les lignes 9, 13 et 14. Son implantation respecte-t-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 bissextile¹. On spécifie pour cela les types *jour* et *mois* ainsi qu'une fonction *estJourDansMois* :

DÉFINITION D'ENSEMBLES

déf *jour* = {1 ... 31}

déf *mois* = {1 ... 12}

SPÉCIFICATION *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.

Exemples :

1. (*estJourDansMois* 28 1) =
2. (*estJourDansMois* 31 4) =
3. (*estJourDansMois* 18 13) n'a pas de sens puisque
4. (*estJourDansMois* 0 4) n'a pas de sens puisque

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

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

DÉFINITION DE TYPES

```
type jour = .... (* restreint à ..... *) ;;
```

```
type mois = .... (* restreint à ..... *) ;;
```

Q2. Compléter la réalisation suivante de la fonction *estJourDansMois* :

RÉALISATION *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.

Implémentation 1

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.

Implémentation 2**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'interalle $\{0 \dots 11\}$. 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.

DÉFINITION D'UN ENSEMBLE

déf *heure* = $\{0 \dots 11\}$

déf *minute* = $\{0 \dots 59\}$

déf *seconde* = $\{0 \dots 59\}$

déf *meridien* = $\{AM, PM\}$

déf *horaire* = *heure* × *minute* × *seconde* × *meridien*

Exemples

1. Le quadruplet $(3, 0, 0, PM)$ représente l'heure exprimée habituellement par 3 : 00 : 00 PM.
2. Le quadruplet $(2, 25, 30, AM)$ représente l'heure 2 : 25 : 30 AM.
3. Notez que $(0, 0, 0, AM)$ correspond à minuit et que $(0, 0, 0, PM)$ correspond à midi.

Incrémenter l'horaire d'une seconde : On considère les fonctions suivantes :

SPÉCIFICATION *incrémenter d'une seconde*

Profil $inc_hor : \text{horaire} \rightarrow \text{horaire}$

Sémantique : $inc_hor(h)$ est l'horaire qui suit d'une seconde l'horaire h

Exemples :

1. $inc_hor(2, 25, 30, \text{AM}) = (2, 25, 31, \text{AM})$
2. $inc_hor(11, 59, 59, \text{PM}) = (0, 0, 0, \text{AM})$

SPÉCIFICATION

Profil $ajout5sec : \text{horaire} \rightarrow \text{horaire}$

Sémantique : $ajout5sec(h)$ est l'horaire qui vient 5 secondes après h

- Q1. Implanter 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 $ajout5sec$.

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 les fonctions de construction et de sélection associées. 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 :

DÉFINITION D'UN ENSEMBLE

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

- Q1. Implanter le type *fraction*.

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

Exemples

1. `frac 21 60` construit le couple correspondant à la réduction de la fraction $\frac{21}{60}$:
`frac 21 60 = (7,20)`
2. `frac 42 120 = (7,20)`
3. `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 :

Profil `pgcd : Z* → Z* → Z*`

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

SPÉCIFICATION

Profil `frac :`

Sémantique :

RÉALISATION

Algorithme :

Implémentation `/!\ conditions utilisation :`

```
let frac (n : ....) (d : ....) : ..... =
.....
```

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 :

Profil `sqrt : R+ → 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 le type *figure* suivant :

DÉFINITION D'UN ENSEMBLE

$$\begin{aligned} \text{déf } \textit{figure} = & \{ \text{TRIANGLE}(t) \mid t \in \dots\dots\dots \} \cup \\ & \{ \text{CERCLE}(\cdot) \mid \dots\dots\dots \} \cup \\ & \{ \dots\dots\dots \mid \dots\dots\dots \} \end{aligned}$$

Q4.

- Que représente le terme `TRIANGLE` dans cette définition ?
- Qu'est-ce-que t ? Quel est son type ? Comment appelle-t-on un tel type ?
- Compléter la spécification du type *figure*.
- Donner une implantation de ce type.

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

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

INDICATION utiliser la composition conditionnelle sous forme de filtrage.

2.6 **TD2** Relations sur des intervalles d'entiers

On caractérise un intervalle **fermé** d'entiers par la donnée de ses bornes inférieure et supérieure.

DÉFINITION D'UN ENSEMBLE *intervalle d'entiers*

déf *Intervalle* = $\mathbb{Z} \times \mathbb{Z}$

Sémantique : 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 *Prédicats sur les intervalles*

Profil *dans* : $\mathbb{Z} \rightarrow \textit{intervalle} \rightarrow \mathbb{B}$

Sémantique : $(\textit{dans } x \ i) = \textit{vrai} \Leftrightarrow x \in i$

Profil *précède* : $\mathbb{Z} \rightarrow \text{intervalle} \rightarrow \mathbb{B}$

Sémantique : (*précède* x i) si et seulement si x est inférieur aux entiers de i

Profil *suit* : $\mathbb{Z} \rightarrow \text{intervalle} \rightarrow \mathbb{B}$

Sémantique : (*suit* x i) si et seulement si x est 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 par des exemples.
- Q3. Réaliser les fonctions *precede*, *dans* et *suit*.

2.6.2 Relations entre intervalles

- Q4. Compléter la spécification de la fonction suivante :

SPÉCIFICATION *relations sur les intervalles*

Profil *coteAcote* : $\text{intervalle} \rightarrow \text{intervalle} \rightarrow \mathbb{B}$

Sémantique : *coteAcote* (i_1, i_2) si et seulement si i_1 et i_2 sont contigus sans se toucher.

Exemples :

- 1. (*coteAcote* (1,3) (4,10)) = vrai
- 2. (*coteAcote* (2,8) (-1,1)) = vrai
- 3. (*coteAcote* (1,3) (3,4)) =
- 4. (*coteAcote* (2,8) (3,4)) =
- 5. cas où les intervalles ont une intersection vide : (*coteAcote*

- Q5. Implanter la fonction *coteAcote* sans utiliser d'expressions conditionnelles ; donner des exemples d'utilisation pertinents.

SPÉCIFICATION *relations sur les intervalles*

Profil *chevauche* : $\text{intervalle} \rightarrow \text{intervalle} \rightarrow \mathbb{B}$

Sémantique : (*chevauche* i_1 i_2) 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. Implanter la fonction *chevauche* sans utiliser d'expressions conditionnelles ; 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. Compléter la spécification des fonctions *codeCar* et *decodeCar* :

2.7.1 Version 1

- Q2.** Expliquer brièvement comment procéder à la main pour coder ou décoder un caractère.
- Q3.** Donner la réalisation de chacune des deux fonctions *codeCar* et *decodeCar*. Faire une composition conditionnelle sous forme d'expressions conditionnelles imbriquées.
- Q4.** Proposer une autre réalisation 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 :

$$\begin{array}{ccc}
 l & \xrightarrow{\text{codeCar}} & l' \\
 l & \xrightarrow{\text{rangCar}} r \xrightarrow{\text{plus3}} r' \xrightarrow{\text{iemeCar}} & l'
 \end{array}$$

- 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 :

DÉFINITION D'UN ENSEMBLE *naturels entre 1 et 26*

déf $\text{nat1_26} = \{1 \dots 26\}$

SPÉCIFICATION

Profil $\text{plus3} : \text{nat1_26} \rightarrow \text{nat1_26}$

Sémantique : $\text{plus3}(r)$ décale le rang r de 3 de manière circulaire dans $\{1 \dots 26\}$

Profil $\text{rangCar} : \text{majuscule} \rightarrow \text{nat1_26}$

Sémantique : $\text{rangCar}(l)$ est le rang dans l'alphabet de la lettre l donnée

Profil $\text{iemeCar} : \text{nat1_26} \rightarrow \text{majuscule}$

Sémantique : $\text{iemeCar}(r)$ est la lettre de rang r dans l'alphabet

- Q5.** Compléter la spécification par des exemples et donner la réalisation de la fonction codeCar en utilisant l'idée ci-avant.
- Q6.** Donner les implantations du type nat1_26 et de la fonction plus3 .
- Q7.** 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 $\{1 \dots 25\}$.

- Q8.** Spécifier les fonctions généralisées de codage codeCarGen et de décodage decodeCarGen , puis donner leur réalisation en utilisant les fonctions intermédiaires adéquates.
- Q9.** Ré-implanter codeCar en utilisant uniquement codeCarGen .

2.8 **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 premier niveau, les pièces sont réparties selon la matière dans laquelle elles sont fabriquées :
déf $matiere = \{ZINC, PLOMB, FER, CUIVRE, ALU, CARBONE, AUTRES\}$
- À un deuxième niveau, on différencie les pièces d'une même matière par un entier entre 0 et 999.
déf $numero = \{0 \dots 999\}$

Une référence 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

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

2.8.1 Types associés à la notion de référence

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

déf $reference = matiere \times numero$

déf $nom = majuscule \times majuscule \times chiffre \times chiffre \times chiffre$

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. Donner les implantations des types *matiere*, *numero*, *reference* et *nom*.

2.8.2 Fonctions associées à la notion de référence

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

SPÉCIFICATION *conversion d'un nom en référence*

Profil $nom_en_ref : nom \rightarrow reference$

Sémantique : $nom_en_ref(n)$ est la valeur de type *reference* associée au nom n .

Exemples :

1. $nom_en_ref('C','A','0','5','3') = (CARBONE,53)$

2. $nom_en_ref('A','L','0','5','3') = (ALU,53)$

Pour réaliser la fonction *nom_en_ref*, on introduit les fonctions *chiffreVbase10* et *cc_en_mat* :

déf $base10 = \{0, \dots, 9\}$

SPÉCIFICATION**Profil** $chiffreVbase10 : chiffre \rightarrow base10$ **Sémantique** : $chiffreVbase10(c)$ est l'entier associé au caractère décimal c .**Exemples** :

1. $chiffreVbase10('2') = 2$
2. $chiffreVbase10('0') = 0$

Profil $cc_en_mat : caractère \times caractère \rightarrow matiere$ **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.**Exemples** :

1. $cc_en_mat('C', 'A') = \text{CARBONE}$
2. $cc_en_mat('C', 'U') = \text{CUIVRE}$
3. $cc_en_mat('X', 'P') = \text{AUTRES}$
4. $cc_en_mat('B', 'Z') = \text{AUTRES}$

Q2. Donner la réalisation de la fonction nom_en_ref en utilisant cc_en_mat et $chiffreVbase10$.**Q3.** Donner la réalisation de la fonction $chiffreVbase10$.**RÉALISATION****Algorithme 1** : composition conditionnelle, sous forme d'expressions conditionnelles imbriquées.**Algorithme 2** : composition conditionnelle, sous forme de filtrageNB : vous avez vu (ou vous verrez) en TP une implantation de la fonction $chiffreVbase10$ qui n'utilise pas d'expression conditionnelle.**Q4.** Réaliser la fonction cc_en_mat .**2.8.3 Comparaison de références : relation d'ordre sur les types**

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

L'ordre caml implicite entre deux valeurs d'un type somme est défini par l'ordre entre les constructeurs du type (et si les valeurs partagent le même constructeur par l'ordre sur l'éventuel 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. Implanter la fonction suivante :

SPÉCIFICATION

Profil $inf_ref : reference \rightarrow reference \rightarrow \mathbb{B}$

Sémantique : $(inf_ref\ r_1\ r_2)$ vaut vrai si et seulement si r_1 est avant r_2 suivant *ordrebis*.

Exemples :

1. $(inf_ref\ (Carbone, 13)\ (Zinc, 45)) = vrai$
2. $(inf_ref\ (PLOMB, 12)\ (ALU, 943)) = vrai$

Chapitre 3

Annales

Sommaire

3.1	Partiel 14–15 TD3 « un dîner presque parfait »	25
3.1.1	Modélisation	25
3.1.2	Coûts des éléments du repas	26
3.1.3	À table	26
3.2	Partiel 13–14 TD3 colorimétrie	27
3.2.1	Colors	28
3.2.2	The grey color	28
3.2.3	Barbouillons	28
3.2.4	Grey and other colors	29

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

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

Q6. (2pt)

- a) Quel est la nature de D et F ?
- b) Implémenter $fromOUdess$ en OCAML
- c) Quel est le profil (ensemble de départ \rightarrow ensemble d'arrivée¹) de D ? De F ?

1

Un repas est implémenté en OCAML de la manière suivante :

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

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 :

Formule₁ : une entrée et un plat,

Formule₂ : un plat et un dessert ou un fromage,

Formule₃ : 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 primaires² : rouge, vert et bleu.

1. ou, plus précisément, *domaine* \rightarrow *codomaine*

2. en synthèse additive

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} *)
type tripletRGB = .....
```

Les couleurs primaires ont une intensité maximale pour leur propre composante, minimale pour les autres ; le noir est défini¹ comme l'absence de couleur ; le blanc est défini¹ comme le mélange à intensité maximale des trois couleurs primaires.

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 Barbouillons³

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

SPÉCIFICATION

Profil $barbouille : tripletRVB \times tripletRVB \rightarrow tripletRVB$

Sémantique : $barbouille(t_1, t_2)$ est la couleur obtenue en mélangeant t_1 et t_2

Exemples :

- soit $jaune = (255, 255, 0)$,
 $barbouille(\text{rouge}, \text{vert}) = barbouille(\text{vert}, \text{rouge}) = jaune$,
- $barbouille(\text{bleu}, \text{jaune}) = barbouille(\text{jaune}, \text{bleu}) = blanc$

3. de *barbouille*, n.f. fam. : peinture de qualité médiocre

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 :

DÉFINITION D’UN ENSEMBLE

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

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 *complémentaire* 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 celà, on spécifie une fonction de conversion des couleurs vers les triplets RVB :

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 $barbouilleC : couleur \times couleur \rightarrow couleur$

Sémantique : $barbouilleC(c_1, c_2)$ est la couleur obtenue en mélangeant c_1 et c_2

Deuxième partie

DÉFINITIONS RÉCURSIVES

Chapitre 4

Types récurifs

Sommaire

4.1	TD4 Puissances d'entiers	32
4.1.1	Définition des types	33
4.1.2	Racine et l'exposant d'une valeur de type <i>puisrec</i>	33
4.1.3	Conversions d'un type vers un autre	34
4.1.4	Produit de deux puissances	35
4.2	TD5 Séq. construites par la droite	36
4.2.1	Type récurif « séquence d'entiers »	36
4.3	TD5 Séq. construites par la gauche	36
4.3.1	Type « séquence d'entiers »	36
4.3.2	Dernier entier	36
4.3.3	Premier élément	37
4.3.4	Séquence de caractères	37
4.3.5	Séquences de couples d'entiers	37
4.3.6	Généralisation : liste de type quelconque	38
4.4	TD6 Le type séquence prédéfini d'OCAML	38
4.5	TD6 Séquences	39
4.6	TD6 Séq. de séq. ou de nuplets	40

4.1 **TD4** Puissances d'entiers

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

- x , que nous appellerons racine, est un entier premier > 1
- p , que nous appellerons exposant, est un entier ≥ 1

4.1.1 Définition des types

Nous utiliserons trois représentations différentes définies par les ensembles *puis*, *puiscple* (un type produit) et *puisrec* (un type récursif).

DÉFINITION D'UN ENSEMBLE

- déf* $puis = \mathbb{N}^* - \{1\}$
- déf* $rac = \{x \in \mathbb{N} \text{ tel que } x \text{ premier}, x > 1\}$
- déf* $exp = \mathbb{N}^*$
- déf* $puiscple = rac \times exp$
- déf* $puisrec = \{R(x) \text{ tel que } x \in rac\} \cup \{P(p) \text{ tel que } p \in puisrec\}$

Exemples de représentations dans les 3 types

Valeurs	Représentations dans les types		
Expressions	<i>puis</i>	<i>puiscple</i>	<i>puisrec</i>
3^1	3	(3,1)	R(3)
3^3	27	(3,3)	P(P(R(3)))
3^4	81	(3,4)	P(P(P(R(3))))
		14641=11*1331=121*121	
		3125=5*625=25*125, 343=7*7*7	

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

Q1.

- Donner les représentations dans les trois types de puissances des constantes suivantes : $5^1, 5^2, 5^5, 7^3, 11^4, 23$.
- Donner en OCAML l'implantation des types *puis*, *rac*, *exp*, *puiscple* et *puisrec*.
- Définir en OCAML les constantes correspondant aux puissances $5^1, 5^2, 5^5, 7^3, 11^4, 23$ dans les types *puis*, *puiscple* et *puisrec*.

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

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

SPÉCIFICATION

Profil $racine : puisrec \rightarrow rac$
Sémantique : $racine(rp)$ retourne la racine de rp

Exemples :

- $racine (R(3)) = .$
- $racine (P(P(P(R(3))))) = .$
- $racine (R(7)) = .$
- $racine (P(P(R(7)))) = .$

SPÉCIFICATION

Profil $exposant : \dots\dots\dots$
Sémantique : $exposant(rp)$ retourne l'exposant de rp

Exemples :

- $exposant (R(3)) = .$
- $exposant (P(P(P(R(3))))) = .$
- $exposant (R(7)) = .$
- $exposant (P(P(R(7)))) = .$

Q3. En se basant sur la structure récursive du type *puisrec* donner les définitions récursives des fonctions racine et exposant.

Définition récursive de la fonction par des équations

- (1) $racine(R(rp)) = ..$
 (2) $racine(P(rp)) =$
 (1bis) $exposant(.....) = .$
 (2bis) $exposant(.....) =$

On va définir une mesure c'est-à-dire une fonction permettant de démontrer la terminaison d'une définition récursive de fonction.

- Q4.**
- Que peut-on dire des arguments d'une fonction mesure ?
 - Rappeler les propriétés que doit posséder une mesure.

On propose la définition suivante pour la fonction : $Mesure(rp) \stackrel{def}{=} \text{nombre de } P \text{ dans } rp = \text{exposant}$

- Q5.** Montrer que $\forall rp \in \text{puisrec}$, l'évaluation de *racine* (*rp*) termine.
Q6. Donner l'implantation OCAML de *racine* et *exposant*.

4.1.3 Conversions d'un type vers un autre

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

SPÉCIFICATION

Profil $puisrec_to_puisuple : puisrec \rightarrow puisuple$

Sémantique : retourne son argument converti dans le type *puisuple*

Exemples :

1. $puisrec_to_puisuple (R(5)) = (5, 1)$
2. $puisrec_to_puisuple (P(P(R(11)))) = (11, 3)$

Profil $puisuple_to_puisrec : puisuple \rightarrow puisrec$

Sémantique : retourne son argument converti dans le type *puisrec*

Exemples :

1. $puisuple_to_puisrec ((5, 1)) = R(5)$
2. $puisuple_to_puisrec (11, 3) = P(P(R(11)))$

Profil $\text{puisrec_to_puis} : \text{puisrec} \rightarrow \text{puis}$

Sémantique : retourne son argument converti dans le type *puis*

Exemples :

1. $\text{puisrec } P(P(R(3))) = 27$

Profil $\text{puiscple_to_puis} : \text{puiscple} \rightarrow \text{puis}$

Sémantique : retourne son argument converti dans le type *puisrec*

Exemples :

1. $\text{puiscple } (11,3) = 1331$

Q7. Donner une réalisation de *puisrec_to_puiscple* en utilisant racine et exposant. Proposer une autre implémentation plus efficace.

Q8. Donner une réalisation récursive de *puiscple_to_puisrec*.

Q9. Donner une réalisation récursive de *puisrec_to_puis*.

Q10. Donner une réalisation récursive de *puiscple_to_puis*.

4.1.4 Produit de deux puissances

On définit deux fonctions calculant le produit de deux puissances telles qu'elles ont été définies au paragraphe 4.1.1.

SPÉCIFICATION

Profil $\text{produit_puiscple} : \text{puiscple} \rightarrow \text{puiscple} \rightarrow \text{puiscple} \times \mathbb{B}$

Sémantique : $\text{produit_puiscple}(x,y)$ est le produit des deux puissances données en argument.

Précondition : x et y ont la même racine.

Le booléen est vrai si la précondition est vérifiée et faux sinon, dans le cas où il est faux, le résultat de type *puiscple* n'a pas de sens.

Profil $\text{produit_puisrec} : \text{puisrec} \rightarrow \text{puisrec} \rightarrow \text{puisrec} \times \mathbb{B}$

Sémantique : $\text{produit_puisrec}(x,y)$ est le produit des deux puissances données en argument.

Précondition : x et y ont la même racine.

Le booléen est vrai si la précondition est vérifiée et faux sinon, dans le cas où il est faux, le résultat de type *puisrec* n'a pas de sens.

Q11. Pourquoi le produit n'a-t-il de sens que si les deux arguments ont même racine ?

Q12. Donner une réalisation de *produit_puiscple*.

Q13. Donner une réalisation de *produit_puisrec*.

4.2 **TD5** Séquences construites par la droite

4.2.1 Type récurif « séquence d'entiers »

On donne la définition du type `seqdEnt`, des séquences construites par ajout à droite d'un élément.

DÉFINITION D'UN ENSEMBLE

déf $seqdEnt = NILdEnt \cup \{ ConsdEnt(prefixe, e) \mid e \in \mathbb{Z}, prefixe \in seqdEnt \}$

- Q1. Donner l'implantation OCAML de cet ensemble.
- Q2. Définir en OCAML les constantes `cst1d` correspondant à la séquence ne comportant que l'élément 5 et `cst2d` définissant la séquence formée des éléments 6, -3 et 7 dans cet ordre.
- Q3. Spécifier et implanter une fonction qui, étant donné un $s \in seqdEnt$, retourne le nombre d'éléments de s .

4.3 **TD5** Séquences construites par la gauche

4.3.1 Type « séquence d'entiers »

On donne la définition des séquences d'entiers construites par ajout à gauche d'un élément :

DÉFINITION D'UN ENSEMBLE

déf $seqE = NilE \cup \{ ConsE(e, suffixe) \mid e \in \mathbb{Z}, suffixe \in seqE \}$

- Q1. Donner l'implantation OCAML de cet ensemble.
- Q2. Définir en OCAML les constantes `cst1g` correspondant à la séquence ne comportant que l'élément 5 et `cst2g` définissant la séquence formée des éléments 6, -3 et 7 dans cet ordre.

4.3.2 Dernier entier

version 1

On veut définir une fonction qui donne l'entier le plus à droite d'une séquence d'entiers construite par la gauche, et on propose la spécification suivante :

SPÉCIFICATION

Profil $dernier_entier : seqE \rightarrow \mathbb{Z}$

Sémantique : $dernier_entier(s)$ est le dernier entier de la séquence s

- Q3. En quoi cette spécification est-elle incorrecte ? Quelle valeur faut-il exclure ?

- Q4.** Donner les équations définissant la fonction `dernier_entier` en vous basant sur la structure récursive du type du paramètre, et en respectant la précondition.
- Q5.** Définir une fonction `mesure` permettant de démontrer la terminaison de la fonction.
- Q6.** Implanter `dernier_entier` en OCAML. Quel message donnera l'interpréteur à l'issue de l'évaluation de cette implantation ?

version 2

On reprend la fonction `dernier_entier` pour laquelle on voudrait une spécification sans précondition et par conséquent sans `warning` lors de l'implantation OCAML.

On propose le profil suivant :

SPÉCIFICATION

Profil $dernier_entier : seqE \rightarrow \mathbb{Z} \times \mathbb{B}$

Sémantique : `dernier_entier (s)` donne un entier e et un booléen b . Si b est vrai, e est l'entier le plus à droite de s , sinon e n'a aucun sens

- Q7.** Compléter la spécification par des exemples d'utilisation de la fonction.
- Q8.** Réaliser cette fonction (équations récursives, terminaison, implantation).

4.3.3 Premier élément

- Q9.** Spécifier et réaliser une fonction `premier_entier` qui retourne le premier élément d'une séquence d'entiers.

4.3.4 Séquence de caractères

Définir le type « séquence de caractères » et réaliser la fonction `dernier_car`.

4.3.5 Séquences de couples d'entiers

On veut définir le type `point` qui représente les coordonnées d'un point dans les deux axes d'un plan et le type séquence de points.

DÉFINITION D'UN ENSEMBLE


déf $point = \{(abs, ord) \mid abs, ord \in \mathbb{R}\}$

déf $seqPoint = \{NilP\} \cup \{ConsP(p, s) \mid p \in point, s \in seqPoint\}$

- Q10.** Spécifier puis réaliser la fonction `dernier_point` qui donne le point le plus à droite d'une séquence de points.

4.3.6 Généralisation : liste de type quelconque

Q11.

1. Définir *Elt* (« élément ») comme une union des entiers, des caractères et des points.
 -  En informatique, si on utilise des langages au typage strict comme OCAML, on ne peut pas mélanger des torchons et des serviettes.
2. Définir *seqElt* comme une séquence d'éléments.
3. Définir une fonction générique *dernier_elt* recevant un paramètre de type *seqElt* et retournant un résultat de type *Elt*.
4. Donner des exemples d'utilisation de la fonction.

4.4 [TD6](#) Le type séquence prédéfini d'OCAML

OCAML prédéfinit le type $seq(\alpha)$ des séquences d'éléments de type α construites par la gauche : `'a list`, ainsi que ses deux constructeurs (utilisables dans un filtrage `match ...with`) `[]` et `--:--`

1. Le symbole `[]` représente le constructeur constant « séquence vide d'éléments » (équivalent des constructeurs constants `Nil` . . . précédents).
2. Le symbole `--:--` représente le constructeur (infixe) d'ajout d'un élément à gauche d'une séquence (équivalent des constructeurs `Cons` . . . précédents).

Par exemple, une séquence composée des trois entiers 8, 5 et 12 dans cet ordre, est de type OCAML `int list` et est définie par l'expression `8::(5::(12::[]))`.

Pour plus de commodité, OCAML accepte aussi la notation synonyme `[8 ; 5 ; 12]` y compris pour un filtrage.

Le type `'a list` représente le type « séquence d'éléments de même type », `'a` étant alors un moyen de désigner le type commun à tous les éléments. La notation ensembliste correspondante est $seq(\alpha)$.

Autres exemples :

- `char list list` est le type « séquence dont chaque élément est de type `char list` » ; notation ensembliste : $seq(seq(caractère))$
- `('a list , 'b list)` définit le type « couple composé d'une séquence d'éléments d'un type quelconque et d'une séquence d'éléments d'un type quelconque éventuellement différent de celui des éléments de la première liste » ; notation ensembliste : $seq(\alpha) \times seq(\beta)$.

Q1. Compléter la définition de la fonction *dernier_elt* ci-dessous :

SPÉCIFICATION

Profil *dernier_elt* :

Sémantique : *dernier_elt*(*s*) est l'élément le plus à droite de la séquence *s*.

Exemples :

1. $\forall e \in \alpha, \text{dernier_elt}(\dots) = e$
2. $\text{dernier_elt}(\dots) = 5.3$
3. $\text{dernier_elt}('a' :: 'b' :: 'c' :: []) = \text{dernier_elt} \dots = \dots$
4. $\text{dernier_elt}([1;2;3];[3;8;7];[];[123]) = \dots$

RÉALISATION

Définition récursive de la fonction par des équations

Voir exercices précédents

Terminaison *Voir exercices précédents*

Implémentation

4.5 TD6 Séquences

Rappels notations OCAML. La séquence vide est notée [], l'ajout à gauche est noté $_{::}$, la concaténation est notée $(_ @ _)$.

Les fonctions donnant respectivement le premier élément et la séquence privée de son premier élément sont prédéfinies en OCAML dans le module `List : List.hd` et `List.tl` :

- `List.hd : seq(α) {[]} → α`
- `List.tl : seq(α) {[]} → seq(α)`

N°	EXPRESSION OCAML	TYPE		EVALUATION OCAML
		ensemble	↔ 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'; '!']

4.6 **TD6** Séquences de séquences ou de nuplets

N°	EXPRESSION OCAML	TYPE		EVALUATION OCAML
		ensemble	↔ OCAML	
1	hd(tl [[4]; [5]; [6]])	↔
2	(0, '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 5

Fonctions récursives sur les séquences

Sommaire

5.1	TD6	Un élément apparaît-il dans une séq.	42
5.2	TD6	Nombre d'occ. d'un élément	43
5.3	TD6	Maximum de n entiers	43
5.4	TD6	Suppression des espaces	44
5.5	TD6	Plus de 'a' que de 'e' ?	45
5.6	TD7	Jouons aux cartes	46
5.6.1		Valeur d'un joker	46
5.6.2		Valeur d'une petite	46
5.6.3		Valeur d'un honneur	46
5.6.4		Valeur d'une main	47
5.7	TD7	Ordre sur des mots ?	47
5.8	TD7	Séq. des valeurs cumulées	47
5.8.1		Réalisation avec découpage et construction de séquence par la droite	47
5.8.2		Découpage à droite de listes construites par la gauche **	48
5.8.3		Découpage à gauche et listes construites par la gauche *	49
5.8.4		Réalisation directe sans fonction intermédiaire *	50
5.9	TD7	*** Le compte est bon	50
5.10	TD8	Quelques tris	51
5.10.1		Tri par insertion	51
5.10.2		Tri par sélection du minimum	51
5.10.3		Tri par pivot/partition (« quicksort »)	51
5.11	TD8	Anagrammes	52
5.11.1		Première réalisation : deux fonctions intermédiaires	52
5.11.2		Seconde réalisation : regroupement des fonctions intermédiaires	52
5.12	TD8	Concaténation de deux séq.	52
5.13	TD8	Préfixe d'une séquence	53

5.1 **TD6** Un élément apparaît-il dans une séquence ?

On cherche à savoir si un certain élément appartient à une séquence. On commence par écrire une fonction qui indique si un caractère particulier, par exemple ' e ', appartient à un texte.

Q1. Implanter l'ensemble *seq* (caractère), appelé *texte*.

La fonction *app_e* est spécifiée suit :

SPÉCIFICATION

Profil $app_e : \text{texte} \rightarrow \mathbb{B}$

Sémantique : *app_e* (t) vaut vrai si et seulement si le caractère ' e ' ∈ t

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

Q3. On veut montrer la terminaison de la fonction récursive *app_e*. Pour cela on définit une fonction appelée *mesure_ape*.

- Que peut-on dire des arguments d'une telle fonction mesure ?
- Rappeler les propriétés que doit posséder une mesure
- On propose la définition suivante pour la fonction *mesure_ape* :
 $mesure_ape(s) \stackrel{def}{=} |s|$, où $|_$ est la fonction mathématique « cardinal », s étant vue ici comme un ensemble de ::
- Montrer que $\forall s \in \text{texte}$, l'évaluation de *app_e* (s) termine

Q4. Implanter la fonction *app_e* de trois manières différentes :

RÉALISATION *app_e_v1*

Algorithme : *filtrage et composition conditionnelle*

RÉALISATION *app_e_v2*

Algorithme : *filtrage et expressions booléennes.*

RÉALISATION *app_e_v3*

Algorithme : * *uniquement des expressions booléennes.*

Généralisons, d'abord à un caractère quelconque.

Rappel Un prédicat est une fonction dont le codomaine est \mathbb{B} , c'est-à-dire dont le type est de la forme : $\dots \rightarrow \mathbb{B}$.

Q5. Définir (spécification, réalisation avec équations récursives et terminaison, implémentation) le prédicat *appCar* qui teste si un certain caractère donné appartient au texte donné.

Généralisons encore, à une séquence d'éléments quelconques.

Q6. Définir le prédicat *app* qui teste si un élément donné appartient à une séquence d'éléments donnée.

Voici quelques exemples d'utilisation de la fonction *app* :

- $app\ 2\ [2; 1; 3] = \text{vrai}$
- $app\ 'm'\ ['e'; 't'; 'h'] = \text{faux}$
- $app\ (1,2)\ [(2,3); (1,2); (2,1)] = \text{vrai}$

Revenons à la fonction *app_e*.

Q7. Donner une implantation non récursive de la fonction *app_e* à l'aide de la fonction *app*.

5.2 TD6 Nombre d'occurrences d'un élément

On souhaite définir une fonction *nbOccZero* qui calcule le nombre d'occurrences de l'entier 0 dans une séquence d'entiers.

- Q1.** Spécifier la fonction *nbOccZero*. On Donnera des exemples d'utilisation qui permettront ensuite de tester cette fonction.
- Q2.** Réaliser la fonction *nbOccZero*.

On veut généraliser la fonction *nbOccZero*.

- Q3.** Définir pour cela une fonction *nbOcc* qui calcule le nombre d'occurrences d'un entier donné (spécification, réalisation avec équations récursives, terminaison et implantation).
- Q4.** En déduire une implantation non récursive de la fonction *nbOccZero*.

5.3 TD6 Maximum de n entiers

On considère les fonctions *max2* et *maxN* spécifiées comme suit :

SPÉCIFICATION

Profil $max2 : \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$

Sémantique : $(max2\ e_1\ e_2)$ est le plus grand des entiers e_1 et e_2

Profil $maxN : seq(\mathbb{Z}) \{[\]\} \rightarrow \mathbb{Z}$

Sémantique : s étant une séquence non vide, $maxN(s)$ est le plus grand des entiers de la séquence s

Q1. Compléter les exemples suivants :

- $maxN [0]=$
- $maxN [-123 ; -10]=$
- $maxN [2 ; 3 ; 1 ; 3 ; 1]=$

Q2. L'expression $maxN ([])$ a-t-elle un sens ?

Q3. Décrire la structure récursive (entité(s) de base, constructeur(s), ainsi que leur type) d'une séquence d'entiers non vide. En déduire des équations récursives pour définir la fonction $maxN$.

Q4. Montrer que pour toute séquence non vide s d'entiers, l'évaluation de $maxN(s)$ termine. S'appuyer sur une fonction mathématique $mesure_maxN$ que vous définirez.

Q5. Donner une réalisation en OCAML des fonctions $max2$ et $maxN$.

Q6. Sur le même modèle que $maxN$ définir une fonction $minN$ qui calcule le plus petit élément d'une séquence non vide d'entiers.

On veut maintenant définir une fonction qui calcule le plus grand entier **ET** le plus petit entier d'une séquence d'entiers non vide.

Pour cela, on définit la fonction $minmax : seq(\mathbb{Z}) \{[]\} \rightarrow \mathbb{Z} \times \mathbb{Z}$.

Quelques exemples d'utilisation de la fonction $minmax$:

- $minmax [4] = (4,4)$
- $minmax [3 ; 5 ; 7 ; 1 ; 8 ; 3] = (1,7)$

Q7. Déduire des questions précédentes une implantation non récursive de la fonction $minmax$.

Q8. * Implanter la fonction $minmax$ sans utiliser les fonctions $maxN$ et $minN$ et en ne réalisant qu'un seul parcours de la séquence donnée en argument.

5.4 **TD6** Suppression des espaces d'un texte

Q1. Définir la fonction $supprEsp$ qui supprime les espaces d'un texte (séquence de caractères) donné. Les autres caractères sont préservés, ainsi que leur ordre d'apparition dans le texte.

Par exemple, $supprEsp[' ' ; ' ' ; 'a' ; 'v' ; ' ' ; 'i' ; ' ' ; ' ' ; 's' ; ' '] = ['a' ; 'v' ; ' ' ; 'i' ; ' ' ; 's' ; ' ']$.

Q2. Que faut-il modifier dans la définition précédente pour ne supprimer les espaces qu'au début du texte ?

Par exemple $supprDeb [' ' ; ' ' ; 'a' ; 'v' ; ' ' ; 'i' ; ' ' ; ' ' ; 's' ; ' '] = ['a' ; 'v' ; ' ' ; 'i' ; ' ' ; ' ' ; 's' ; ' ']$

Q3. En proposant un nom évocateur, généraliser la fonction $supprEsp$ (spécification + réalisation) pour qu'elle effectue la suppression d'un caractère quelconque donné.

Q4. En déduire une autre implantation de la fonction $supprEsp$.

Q5. Conclusion : est-il plus difficile d'écrire la fonction généralisée ?

5.5 **TD6** Plus de 'a' que de 'e' ?

Soit la fonction : $plusAE$:

SPÉCIFICATION

Profil $plusAE : \text{texte} \rightarrow \mathbb{B}$

Sémantique : $plusAE(t)$ vaut vrai si et seulement si t comporte strictement plus de caractères 'a' que de caractères 'e'

Q1. Compléter les exemples suivants :

- $plusAE [] = \dots$
- $plusAE ['e'; 'l'; 'e'; 'a'; 'n'; 'o'; 'r'] = \dots$
- $plusAE ['N'; 'a'; 't'; 'h'; 'a'; 'l'; 'i'; 'e'] = \dots$

Pour faciliter la définition de $plusAE$ on introduit la fonction :

SPÉCIFICATION

Profil $nbOcc : \text{car} \rightarrow \text{texte} \rightarrow \mathbb{N}$

Sémantique : $(nbOcc\ c\ t)$ est le nombre d'apparitions du caractère c dans le texte t

Q2. Réaliser la fonction $plusAE$ à l'aide de la fonction $nbOcc$.

Q3. Réaliser la fonction $nbOcc$.

On se propose maintenant d'utiliser une autre fonction, $nbAE$, pour réaliser $plusAE$.

SPÉCIFICATION

Profil $nbAE : \text{seq}(\text{car}) \rightarrow \mathbb{N} \times \mathbb{N}$

Sémantique : Posons $(n_a, n_e) = nbAE(t)$; n_a est le nombre de 'a' et n_e le nombre de 'e' du texte t .

Q4. Réaliser $plusAE$ à l'aide de $nbAE$.

Q5. * Réaliser $nbAE$ sans utiliser $nbOcc$, en réalisant un seul parcours de la séquence argument.

On considère une fonction $diffAE$ qui calcule la différence entre le le nombre de 'a' et le nombre de 'e' d'un texte.

Q6. Après avoir précisé la spécification de $diffAE$, réaliser $plusAE$ à l'aide de $diffAE$ et sans utiliser $nbOcc$.

Q7. Donner une réalisation de $diffAE$.

Q8. Etudier la généralisation du problème à deux lettres. quelconques.

5.6 [TD7](#) Jouons aux cartes

L'objectif de cet exercice est d'expérimenter la définition d'une fonction récursive (partie 2) 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 :

DÉFINITION D'ENSEMBLES

déf *joker* = {JOKROUGE, JOKNOIR}

déf *figure* = {♣, ◇, ♥, ♠}

déf *basse* = {2 ... 10}

déf *petite* = *basse* × *figure*

déf *haute* = {AS, ROI, DAME, VALET}

déf *honneur* = *haute* × *figure*

déf *carte* = {H(*h*) / *h* ∈ *honneur*} ∪ {P(*p*) / *p* ∈ *petite*} ∪ {J(*j*) / *j* ∈ *joker*}

- Q1.** Préciser la nature (énuméré, somme, produit, intervalle) de chacun de ces types, et en donner une réalisation en caml.

5.6.1 Valeur d'un joker

- Q2.** Dans ce jeu, imaginaire, un joker vaut 50 points quelle que soit sa couleur. Donner la définition de la constante VALJOKER.

5.6.2 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, ce qui s'écrit : *valPetite* (9,◇)=9, *valPetite* (9,♠)=9

- Q3.** Donner d'autres exemples d'utilisation de la fonction *valPetite*. Spécifier la fonction *valPetite* puis en donner une réalisation.

5.6.3 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 la fonction *valHonneur*. Donner des exemples d'utilisation de cette fonction. En donner une réalisation. *valHonneur* : *honneur* → int

5.6.4 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 *cstM1* 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 (cstM1)* ? Donner une spécification de la fonction *valMain* puis donner des équations la définissant et enfin donner en une réalisation.

5.7 TD7 Ordre sur des mots ?

On veut comparer des mots selon les critères égalité et inférieur selon l'ordre alphabétique, c'est-à-dire est placé avant dans le dictionnaire.

On représente un mot par une séquence de caractères.

Q1. Définir le type mot.

Q2. Spécifier une fonction à valeur booléenne déterminant si deux mots sont égaux. Donner des équations récursives définissant la fonction puis une réalisation en OCAML.

Q3. Spécifier une fonction à valeur booléenne déterminant si un mot est (strictement) avant un autre mot. Donner des équations récursives définissant la fonction puis une réalisation en OCAML.

5.8 TD7 Séquence des valeurs cumulées

On veut calculer les *valeurs cumulées* d'une séquence d'entiers. On spécifie ainsi une fonction *valCum* :

SPÉCIFICATION

Profil $valCum : seq(\mathbb{Z}) \rightarrow seq(\mathbb{Z})$

Sémantique : $valCum [e_0 ; e_1 ; e_2 ; e_3 ; \dots ; e_n] = [e_0 ; e_0 + e_1 ; e_0 + e_1 + e_2 ; \dots ; e_0 + \dots + e_{n-1}]$.

Exemple : $valCum [5 ; 4 ; 3 ; 2 ; 0 ; 1] = [5 ; 9 ; 12 ; 14 ; 14 ; 15]$

5.8.1 Réalisation avec découpage et construction de séquence par la droite

Nous étudions une première réalisation de la fonction avec un découpage à droite. Nous reprenons la définition du type séquence d'entier du paragraphe 4.2 : *valCum* a ainsi le profil $seqdint \rightarrow seqdint$, le type *seqdint* étant défini ainsi :

```
type seqdint = NILD | ConsD of seqdint * int
```

Nous noterons aussi les séquences entre crochets pour des raisons de lisibilité. Par exemple, $[2;1;3]$ est synonyme de $\text{ConsD}(\text{ConsD}(\text{NILD}, 1), 2), 3)$.

Nous supposerons également l'existence de deux sélecteurs *premierg* et *dernierd* retournant respectivement l'élément à l'extrémité gauche (*premierg*) et droite (*dernierd*) d'une séquence non vide :

SPÉCIFICATION

Profil $\text{premierg} : \text{seq}(\mathbb{Z}) - \{[]\} \rightarrow \mathbb{Z}$
Sémantique : $\text{premierg} [e_0 ; e_1 ; e_2 ; \dots ; e_{n-1}] = e_0$

Exemple : $\text{premierg} [5 ; 4 ; 3 ; 2 ; 0 ; 1] = 5$

Q1.

1. Que vaut $\text{valCum} [2 ; 1 ; 4 ; 6]$?
2. Que vaut $\text{valCum} [2 ; 1 ; 4]$?
3. Remarquons que $[2 ; 1 ; 4 ; 6] = \text{ConsD}([2 ; 1 ; 4], 6)$. Quelles opérations doit-on effectuer pour passer du résultat de $\text{valCum} [2 ; 1 ; 4]$ à celui de $\text{valCum} [2 ; 1 ; 4 ; 6]$?
4. Quelle fonction sur les séquences serait utile pour calculer le dernier élément de la séquence des valeurs cumulées ?
5. Définir (spécification et réalisation) cette fonction.

On s'appuie sur la définition récursive du type `seqd` pour définir la fonction *valCum* par des équations récursives.

Q2. Donner les parties droites des équations :

- $\text{valCum NILD} =$
- $\forall s \in \text{seqd}, e \in \text{int}, \text{valCum consD}(s, e) =$

Q3. On étudie la terminaison de la fonction *valCum*. On propose la définition de la fonction `mesure_vc` (s) = longueur de la séquence s .

1. Quelles sont les propriétés de la fonction `mesure_vc` (s) ?
2. Faire une preuve de terminaison comme habituellement.
3. Est-ce suffisant pour garantir la terminaison de la fonction *valCum* ?

Implantation**5.8.2 Découpage à droite de listes construites par la gauche ****

Le principe du découpage à droite précédent peut être appliqué à des séquences construites classiquement par la gauche, en utilisant les fonctions ou opérateur suivant :

1. Opérateur de concaténation de deux séquences $@$: (ex. : $[1;2]@[3;4] = [1;2;3;4]$)
2. Fonction *dernier* retournant le dernier élément d'une séquence (ex. : *dernier* $[1;2;3] = 3$)
3. Fonction *debut* retournant une copie de la séquence privée de son dernier élément (ex. : *debut* $[1;2;3] = [1;2]$)
4. Ou bien une fonction combinée *debutdernier* retournant un couple (*deb, der*) en un seul parcours récursif de la séquence.

Q4. Réécrire les équations de *valcum* en remplaçant le constructeur `ConsD` par $@$, *dernier* et *debut*. On peut noter que toute séquence *non vide* s vérifie la propriété $s = (\text{debut } s) @ [\text{dernier } s]$.

Q5.

- a) Donner les équations de *debut* et *dernier* et le code OCAML de *debutdernier*.
- b) Implémenter *debut* et *dernier* non récursivement en utilisant *debutdernier*.

Q6. Implémenter *valcum*.

INDICATION Noter que $@$, *debut* et *dernier* ne sont pas des constructeurs et ne peuvent donc pas être utilisés dans un filtrage ; utiliser une composition conditionnelle pour différencier les 2 cas correspondant aux 2 équations.

On remarque que le somme des éléments est justement le dernier élément des valeurs cumulées.

Q7. Implémenter *valcum* sans utiliser la fonction *somme*.

5.8.3 Découpage à gauche et listes construites par la gauche *

Les listes en OCAML étant construites par la gauche on travaille avec le profil suivant pour *valCum* : $\text{valCum} : \text{int list} \rightarrow \text{int list}$.

Q8.

- a) Quelle est la valeur de *valCum* $[2;1;5;4]$?
- b) Quelle est la valeur de *valCum* $[1;5;4]$?
- c) Remarquons que $[2;1;5;4] = 2 :: [1;5;4]$. Quelle opération doit-on effectuer pour passer du résultat de *valCum* $[1;5;4]$ à celui de *valCum* $[2;1;5;4]$?
- d) Quelle fonction sur les séquences serait utile pour calculer la séquence des valeurs cumulées ?
- e) Définir (spécification et réalisation) cette fonction.

Q9. Donner des équations récursives définissant *valCum* en utilisant *ajouterAchaque*.

Q10. Montrer la terminaison de la fonction *valCum* ainsi définie.

Q11. Donner une implantation des fonctions *ajouterAchaque* et *valCum* en OCAML.

5.8.4 Réalisation directe sans fonction intermédiaire *

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.

Q12.

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 $valCum [1 ; 3 ; 5 ; 7]$?

Q13. Donner les parties droites des équations suivantes :

1. $valCum [] =$
2. $valCum [e] =$
3. $valCum (e_1 : : e_2 : : suite) =$

Q14. Tester l'algorithme proposé en l'appliquant au calcul de $valCum [2 ; 5 ; 10 ; 12]$.

5.9 **TD7** *** Le compte est bon

On étudie une forme simplifiée du jeu "le compte est bon". Étant donné un entier x strictement positif et une séquence s d'entiers strictement positifs, on cherche à exprimer x comme une somme d'éléments de s .

Par exemple, pour $x=20$ et $s=[18 ; 3 ; 1 ; 4 ; 20]$, on a une solution : $x=20$. Pour $x=25$ et $s=[21 ; 3 ; 2 ; 4 ; 2]$, on a deux solutions : $x=21+4$ et $x=21+2+2$. Pour x 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.

Indication : On pourra raisonner en suivant le principe suivant : étant donné une séquence non vide de la forme $e : : suite$, on considère le cas où une solution comporte l'entier e (possible si $e=x$ ou $e<x$) et le cas où la solution ne comporte pas e (dans le cas où $e>x$). Les solutions ne comportant pas l'entier e sont une instance du même problème pour x et $suite$, les solutions avec e sont déduites d'une instance du même problème pour $x-e$ et $suite$.

On considère les quatre fonctions suivantes :

SPÉCIFICATION

Profil $existeSol : seq(\mathbb{Z}) \rightarrow \mathbb{Z} \rightarrow \mathbb{B}$

Sémantique : prédicat qui détermine s'il existe une solution

Profil $nbSol : seq(\mathbb{Z}) \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$

Sémantique : détermine le nombre de solutions

Profil $uneSol : seq(\mathbb{Z}) \rightarrow \mathbb{Z} \rightarrow seq(\mathbb{Z})$

Sémantique : calcule une solution sous forme d'une séquence d'entiers

Profil $lesSol : seq(\mathbb{Z}) \rightarrow \mathbb{Z} \rightarrow seq(seq(\mathbb{Z}))$

Sémantique : calcule toutes les solutions, sous forme d'une séquence de séquences d'entiers

- Q1. Pour chacune des fonctions donner des équations récursives, montrer la terminaison et donner une solution en OCAML.

5.10 TD8 Quelques tris

On étudie une fonction *tri* définie par :

SPÉCIFICATION

Profil $tri : seq(\mathbb{Z}) \rightarrow seq(\mathbb{Z})$

Sémantique : *tri* (*s*) est une séquence formée des éléments de *s* rangés en ordre croissant

5.10.1 Tri par insertion

Considérons une séquence construite à partir d'un premier élément *e* et d'une séquence de fin *s*. L'idée est de trier la partie *s* puis d'insérer l'entier *e* dans la séquence obtenue.

- Q1. Donner des équations définissant la fonction `tri` selon le principe énoncé. Spécifier et réaliser les fonctions intermédiaires nécessaires.

5.10.2 Tri par sélection du minimum

s étant une séquence d'entiers, soit *m* la valeur du plus petit entier de cette séquence, et *s'* la séquence obtenue en privant *s* de cet élément *m*. On obtient une séquence triée en plaçant *m* en tête de la séquence *s'* triée.

- Q2. Donner des équations définissant la fonction `tri` selon le principe énoncé. Spécifier et réaliser les fonctions intermédiaires nécessaires.

5.10.3 Tri par pivot/partition (« quicksort »)

Soit *s* une séquence d'entiers. On la découpe en trois parties : un entier *p*, deux séquences *deb* et *fin* ; *p* est le premier élément de *s* : *deb* est une séquence formée des entiers de *s* inférieurs ou égaux à *p* : *fin* est une séquence formée des entiers de *s* strictement supérieurs à *p*.

Par exemple, pour $s = [7 ; 5 ; 25 ; 3 ; 7 ; 15 ; 4 ; 20]$, $p=7$, $deb=[5 ; 3 ; 7 ; 4]$ et $fin=[25 ; 15 ; 20]$.

On obtient une séquence triée formée des éléments de *s* en concaténant la séquence triée des éléments de *deb*, l'entier *p* et la séquence triée des éléments de *fin*.

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

5.11 **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 « nacre », « arsenelupin » et « paulsernine » et « luisperenna ». Par contre « lasse » et « salle » ne sont pas des anagrammes. Tout mot est anagramme de lui-même.

On étudie une fonction à valeur booléenne, nommée *estAna*, qui détermine s'il est vrai que deux mots donnés sont des anagrammes. Une solution est fondée sur le principe suivant : chaque lettre du premier mot doit appartenir au second et le second ne doit pas en comporter d'autres.

5.11.1 Première réalisation : deux fonctions intermédiaires

- Q1.** Spécifier *estAna*.
- Q2.** Spécifier deux fonctions intermédiaires, l'une nommée *appCar* d'appartenance d'une lettre à un mot et l'autre nommée *supprCar* de suppression d'une lettre dans un mot.
- Q3.** Réaliser *estAna* en utilisant ces fonctions.
- Q4.** Réaliser les fonctions intermédiaires, à moins que cela n'ait déjà été fait dans un autre exercice.

5.11.2 Seconde réalisation : regroupement des fonctions intermédiaires

- Q5.** On définit une fonction *appSuppr* qui regroupe les deux fonctions *appCar* et *supprCar* en une seule. Spécifier la fonction *appSuppr*.
- Q6.** Donner une deuxième réalisation de *estAna* obtenue en utilisant *appSuppr*,
- Q7.** Donner une réalisation de la fonction *appSuppr*,

5.12 **TD8** Concaténation de deux séquences

On connaît 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.** Donner la spécification de la fonction *concat*.

Q2. Donner les parties droites des équations récursives suivantes :

1. $(concat\ []\ []) = ..$

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

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

4. $(concat\ e_1 :: s_1\ s_2) =$

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 $\{1, 3, 4\}$ est insuffisante.

Q4. Expliquer pourquoi la combinaison $\{1, 2, 4\}$ n'est pas minimale.

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

Q6. Donner une réalisation de *concat*.

5.13 TD8 Préfixe d'une séquence

Une séquence, dénotée *pre*, est un préfixe d'une séquence *s* si et seulement si il existe une séquence, dénotée *term* (terminaison), telle que $s = pre @ term$. 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 *lesPréf* qui construit une séquence comportant **tous** les préfixes d'une séquence d'entiers donnée. Exemple : $lesPréf\ [1;2;3] = [[]; [1]; [1;2]; [1;2;3]]$.

Chapitre 6

Annales

Sommaire

6.1	Partiel 14–15 TD8 « gardez la monnaie »	54
6.1.1	Modélisation d'un porte-monnaie	54
6.1.2	Somme d'argent et porte-monnaie	55
6.1.3	Shopping	56
6.1.4	Porte-monnaie et séquences de pièces	56
6.2	Exam 14–15 TD8 décomposition en facteurs premiers	57
6.3	Exam 14–15 TD8 relations binaires	59

6.1 Partiel 2014–2015 [TD8](#) « gardez la monnaie »

Le $km\text{€}^1$ est une unité monétaire, en vogue chez les programmeurs fonctionnels, comportant des pièces de 1, 2, 5 et 10 unités.

1, 2, 5 et 10 sont les *valeurs faciales* des pièces de 1, 2, 5 et 10 $km\text{€}$

6.1.1 Modélisation d'un porte-monnaie

On modélise le contenu d'un porte-monnaie comme un quadruplet (a, b, c, d) où :

- a est le nombre de pièces de 1 $km\text{€}$,
- b est le nombre de pièces de 2 $km\text{€}$,
- c est le nombre de pièces de 5 $km\text{€}$,
- d est le nombre de pièces de 10 $km\text{€}$,

grâce aux types `nat` et `porteMonnaie` :

```
type nat = int (* >= 0 *)
type porteMonnaie = nat * nat * nat * nat
```

1. prononcer « caml »

- Q1.** (0,5pt) Définir la constante `cstPM1` représentant un porte-monnaie contenant deux pièces de 1 *km£*, trois pièces de 2 *km£*, et une pièce de 10 *km£*.

6.1.2 Somme d'argent et porte-monnaie

La *valeur* d'un porte-monnaie est la somme totale d'argent (un entier naturel) qu'il représente.

- Q2.** (0,5pt) Implémenter une fonction *valeur* calculant la valeur d'un porte-monnaie.

La transformation d'un entier naturel en porte-monnaie (opération réciproque de celle de la question précédente) est spécifiée ainsi :

SPÉCIFICATION

Profil $natVpm : \mathbb{N} \rightarrow porteMonnaie$

Sémantique : $natVpm(n)$ est un porte-monnaie dont la valeur est n .

La réalisation de la fonction *natVpm* peut s'effectuer de différentes manières.

- Q3.** (2pt) Étant donné un entier naturel n quelconque, implémenter *natVpm* de deux façons différentes :

- a) `natVpm_simple(n)` est le porte-monnaie de valeur n ne comportant que des pièces de 1 *km£*.
- b) `natVpm_normalise(n)` est le porte-monnaie de valeur n *normalisé*, c'est-à-dire comportant un nombre minimal de pièces (à valeur identique, les ensembles de « petites » pièces sont remplacés par moins de pièces, mais plus « grosses »).

Les fonctions *valeur* et *natVpm* (version simple ou normalisée) sont liées par la relation mathématique suivante :

$$valeur \circ natVpm = id_{\mathbb{N}}$$

où $id_{\mathbb{N}}$ est la fonction identité dans \mathbb{N} . Autrement dit :

$$\forall n \in \mathbb{N}, valeur(natVpm(n)) = n$$

- Q4.** (0,5pt) Donner un exemple d'expression `OCAML` permettant de vérifier cette relation pour un n particulier.

La réciproque de cette relation n'est en général pas vraie :

$$natVpm \circ valeur \neq id_{porteMonnaie}$$

- Q5.** (1,5pt)

- a) Illustrer le fait que $\exists pm \in porteMonnaie / natVpm(valeur(pm)) \neq pm$
- b) Implémenter une fonction *normalise* qui transforme un porte-monnaie en porte-monnaie normalisé.

6.1.3 Shopping

Acheter avec son porte-monnaie consiste à échanger un objet ayant un certain *prix* contre des pièces du porte-monnaie. L'action d'acheter se traduit financièrement par un *débit* du porte-monnaie.

SPÉCIFICATION

Profil $\text{debit} : \text{porteMonnaie} \rightarrow \mathbb{N} \rightarrow \text{porteMonnaie}^1$

Sémantique : (*debit pm p*) est le porte-monnaie résultant de l'achat d'un objet de prix *p* avec le porte-monnaie *pm*.

On supposera que :

- il y a assez d'argent dans *pm* pour pouvoir acheter l'objet,
- le contenu de *pm* permet de faire l'appoint.

Q6. (1,5pt) Implémenter la fonction *debit*.

6.1.4 Porte-monnaie et séquences de pièces

On modélise maintenant un porte-monnaie comme une séquence² de pièces :

```
type piece = Un | Deux | Cinq | Dix
type porteMonnaie2 = Vide | Cons of piece * porteMonnaie2
```

Q7. (1pt)

a) Quelle est la valeur des porte-monnaies suivants :

- Vide
- Cons (Un, Vide)
- Cons (Deux, Cons (Dix, Cons (Deux, Cons (Cinq, Vide))))

b) Définir la constante `cstPM2` représentant un porte-monnaie contenant deux pièces de 1 *km£*, trois pièces de 2 *km£*, et une pièce de 10 *km£*.

Soit la fonction *valeur2* de profil $\text{porteMonnaie2} \rightarrow \mathbb{N}$ calculant la somme totale d'argent représentée par un porte-monnaie.

Q8. (3pt)

- Donner les équations récursives définissant *valeur2*.
- En déduire une implémentation en OCAML de *valeur2*.
- Définir une mesure décroissant strictement entre deux appels récursifs.
- Monter que $\forall pm \in \text{porteMonnaie2}$, l'évaluation de *valeur2(pm)* se termine.

1. ou, si l'on préfère, $\text{porteMonnaie} \times \mathbb{N} \rightarrow \text{porteMonnaie}$

2. mathématiquement ici : un multi-ensemble

- Q9.** (1,5pt) Implémenter une fonction `pm2Vpm` de conversion d'un `porteMonnaie2` en `porteMonnaie`.
- Q10.** (2pt) Question bonus hors barème.
Implémenter une fonction `pmVpm2` de conversion d'un `porteMonnaie` en `porteMonnaie2`.

6.2 Examen 2014–2015 **TD8** décomposition en facteurs premiers

On s'intéresse dans cet exercice aux entiers naturels non nuls (\mathbb{N}^*). En mathématiques, la décomposition en produit de facteurs premiers, consiste à chercher à écrire un entier naturel non nul sous forme d'un produit de nombres premiers. Cette factorisation existe pour tout entier naturel et est toujours unique.

Par exemple, si le nombre donné est 45, la factorisation en nombres premiers est : $3^2 \times 5$, soit $3 \times 3 \times 5$.

Autres exemples :

$$125 = 5 \times 5 \times 5 = 5^3 \quad 360 = 2 \times 2 \times 2 \times 3 \times 3 \times 5 = 2^3 \times 3^2 \times 5 \quad 1001 = 7 \times 11 \times 13$$

Représentation d'un nombre et fonctions de base

Pour représenter un nombre entier par sa décomposition en facteurs premiers, on donne les définitions d'ensemble suivantes.

Définition d'ensembles :

- *premier* = $\{x \in \mathbb{N}^* \setminus \{1\} \mid x \text{ est un nombre premier, c.a.d différent de 1, et divisible uniquement par lui même}\}$
- *exp* = \mathbb{N}^* (exposant d'un facteur premier)
- *facteur* = *premier* \times *exp*
- *nombre* = *seq(facteur)* telle que tous les facteurs de la séquence portent sur des nombres premiers différents deux à deux, et que ces facteurs soient rangés par ordre croissant des nombres premiers. Par convention, la séquence vide représente l'entier 1.

Exemples et contre-exemples :

- le nombre 360 est représenté par la séquence $[(2, 3) ; (3, 2) ; (5, 1)]$;
- le nombre 7 est représenté par la séquence $[(7, 1)]$;
- les trois séquences $[(3, 2) ; (2, 3) ; (5, 1)]$, $[(6, 1)]$ et $[(2, 3) ; (2, 4) ; (3, 3)]$ ne sont pas conformes à la définition du type *nombre*.

- Q1.** (1pt) Donner l'implantation des quatre ensembles ci-dessus sous forme de types Caml (on utilisera le type `list` pour représenter une séquence). Donner l'expression Caml représentant le nombre 120.

On donne les spécifications des trois fonctions suivantes :

- **Profils :**

- $puiss : \mathbb{Z} \rightarrow \mathbb{N} \rightarrow \mathbb{Z}$
- $int_of_facteur : facteur \rightarrow \mathbb{N}^*$
- $int_of_nombre : nombre \rightarrow \mathbb{N}^*$

- **Sémantique :**

- $puiss\ x\ y = x^y$
- $int_of_facteur\ f$ retourne la valeur entière du facteur premier f
- $int_of_nombre\ nb$ retourne la valeur entière du nombre dont la décomposition en facteurs premiers est nb

- **Exemples :**

$$\begin{aligned} & \text{puis } 2\ 4 = 16 \\ & \text{int_of_facteur } (5, 3) = 125 \\ & \text{int_of_nombre } [(2, 3); (3, 2); (5, 1)] = 360 \end{aligned}$$

- Q2.** (3pt) Réaliser en Caml les fonctions `int_of_facteur` et `int_of_nombre` (la fonction `puiss` pourra être utilisée, sa réalisation n'est pas demandée).
- Q3.** (3pt)

1. Ecrivez en CAML la valeur (de type `nombre`) représentant la décomposition en facteurs premiers d'un nombre premier $p > 1$.
2. Spécifier et donner une implantation en Caml des prédicats `estPair` et `estPremier` qui indiquent si un nombre donné par sa décomposition en facteur premier est pair (respectivement premier).

Indication : un nombre pair contient le nombre premier 2 dans le 1er facteur de sa décomposition ; un nombre premier est soit le nombre 1, soit ne possède qu'un facteur premier dans sa décomposition.

Calcul sur les nombres

- Q4.** (3pt) Spécifier et réaliser (**équations de récurrence uniquement**) une fonction *produit* de profil : `nombre → nombre → nombre`, qui retourne le produit de deux nombres donnés. On s'attachera à donner des exemples et/ou des propriétés dans la spécification.

Indication : décomposer récursivement les deux paramètres et tenir compte des propriétés de la séquence de facteurs premiers (unicité des nombres premiers, séquences ordonnées).

Q5. (3pt) Réaliser (implantation en caml uniquement) une fonction *pgcd* de profil : nombre \rightarrow nombre \rightarrow nombre, qui retourne le “plus grand commun diviseur” de deux nombres donnés. On rappelle quelques propriétés et exemples du *pgcd* :

- le *pgcd* de 1 et de tout nombre vaut 1
- le *pgcd* de 16 et 4 vaut 4 :
 $\text{pgcd } [(2, 4)] [(2, 2)] = [(2, 2)]$
- le *pgcd* de 126 et 360 vaut 18 :
 $\text{pgcd } [(2, 1) ; (3, 2) ; (7, 1)] [(2, 3) ; (3 ; 2) ; (5, 1)] = [(2, 1) ; (3, 2)]$

Utilisation de fonctions d’ordre supérieur sur les séquences

Q6. (4pt) Donner les implantations Caml **non récursives (utilisant les schémas d’ordre supérieur donnés en Annexe)** des fonctions suivantes :

1. prédicat *estCarre*, de profil nombre \rightarrow \mathbb{B} , indiquant si un nombre est un carré.
Indication : tout entier supérieur ou égal à 1 est un carré si tous les exposants de sa décomposition en produit de facteurs premiers sont pairs.
2. fonction *carré*, de profil nombre \rightarrow nombre, d’élévation au carré.
3. prédicat *divisible*, de profil nombre \rightarrow premier \rightarrow \mathbb{B} où (divisible nb p) vaut vrai ssi nb est divisible par le nombre premier p.

6.3 Examen 2014–2015 **TD8** relations binaires

On considère dans cet exercice des *relations binaires* définies sur un ensemble donné E . Formellement, une telle relation R est une partie du produit cartésien $E \times E$ ($R \subseteq E \times E$). La relation R peut donc être représentée par l’ensemble des couples (x, y) de $E \times E$ qui la constituent.

On choisit dans la suite de représenter cet ensemble de couples par une séquence, qui sera implémentée en utilisant le type `list` de Caml. Le type *relation* sera donc défini comme suit :

```
type 'a couple = 'a * 'a (* un couple est un element de E x E *)
type 'a relation = 'a couple list (* une relation est un ensemble de couples *)
```

Dans cette définition :

- l’ensemble E est représenté par le type générique `'a` ;
- la liste ne contient pas de doublons : le même couple (x, y) ne pas être présent plus d’une fois dans cette liste (par contre les couples (x, y) et (y, x) peuvent être présents tous les deux) ;
- cette liste de couples (x, y) n’est pas ordonnée.

On donne ci-dessous trois exemples de relations définies sur un ensemble de chaînes de caractères :

```
let r1 = [("Alain", "Michelle"); ("Philippe", "Alain"); ("Philippe", "Michelle")]
```

```
let r2 = [("Alain", "Michelle"); ("Laura", "Philippe"); ("Michelle", "Alain");
          ("Philippe", "Laura")]
```

```
let r3 = [("Alain", "Laura"); ("Laura", "Laura"); ("Philippe", "Alain");
          ("Alain", "Alain"); ("Philippe", "Philippe")]
```

Fonctions de base

- Q1.** (2pt) Ecrire en Caml une fonction *appartient* qui prend en paramètre un couple c et une relation r et qui vaut vrai si et seulement $c \in r$.

Exemple : `appartient ("Laura", "Philippe") r2`

- Q2.** (3pt) Ecrire en Caml une fonction *union* qui prend en paramètre deux relation `rel1` et `rel2` et qui renvoie la relation **union** de `rel1` et `rel2` :

$$\text{union}(\text{rel1}, \text{rel2}) = \{(x, y) \mid (x, y) \in \text{rel1} \vee (x, y) \in \text{rel2}\}$$

Exemple :

```
union r1 r2 = [("Alain", "Michelle"); ("Philippe", "Alain"); ("Philippe", "Michelle");
              ("Laura", "Philippe"); ("Michelle", "Alain"); ("Philippe", "Laura")]
```

Symétrie

Une relation R est dite *symétrique* si et seulement si elle vérifie la propriété suivante :

$$\forall (x, y), (x, y) \in R \Rightarrow (y, x) \in R$$

La relation `r2` est symétrique, les relations `r1` et `r3` ne sont pas symétriques. Notons que la relation “vide” est symétrique.

- Q3.** (3pt) Ecrire en Caml un prédicat *estSymétrique* qui prend en paramètre une relation r et qui vaut vrai si et seulement si r est symétrique.

Indication : une solution possible consiste à vérifier que pour chaque couple (x, y) de la relation r , alors le couple (y, x) appartient également à r . Il peut être nécessaire pour cela d'utiliser une *fonction auxiliaire*.

Réflexivité

Une relation R est dite *réflexive*² si et seulement si elle vérifie la propriété suivante :

$$\forall (x, y), (x, y) \in R \Rightarrow (x, x) \in R \wedge (y, y) \in R$$

2. plus précisément *quasi-réflexive*

La relation `r3` est réflexive, les relations `r1` et `r2` ne sont pas réflexives. Notons que la relation “vide” est réflexive.

- Q4.** (2pt) Ecrire en Caml un prédicat *estRéflexive* qui prend en paramètre une relation `r` et qui vaut vrai si et seulement si `r` est réflexive.

Indication : on peut écrire cette fonction en se basant sur le même raisonnement que pour *estSymétrique*.

Transitivité

Enfin, une relation `R` est dite *transitive* si et seulement si elle vérifie la propriété suivante :

$$\forall (x, y, z), ((x, y) \in R \wedge (y, z) \in R) \Rightarrow (x, z) \in R$$

La relation `r1` est transitive, les relations `r2` et `r3` ne sont pas transitives. Notons que la relation “vide” est transitive.

- Q5.** (4pt) Ecrire en Caml un prédicat *estTransitive* qui prend en paramètre une relation `r` et qui vaut vrai si et seulement si `r` est transitive.

Indication : pour écrire cette fonction on peut utiliser l’algorithme qui consiste à vérifier que, pour chaque couple (x_0, y_0) de `r` :

1. soit `ly0`, la liste des couples de la forme (y_0, z) de `r` ;
2. pour chaque élément (y_0, z) de `ly0`, vérifier que (x_0, z) appartient à `r`

Pour implémenter cet algorithme, la fonction prédéfinie `List.filter` rappelée en Annexe peut être utilisée (sans la ré-écrire) pour construire la liste `ly0`.

Troisième partie

ORDRE SUPÉRIEUR

Chapitre 7

Utilisation/conception de fonctions d'ordre supérieur

Sommaire

7.1	TD9 Fonctions d'ordre sup. simples	63
7.2	TD9 Composition, fonction locale	64
7.3	TD9 Utilisation des fonctions d'ordre sup. <i>map</i> et <i>fold</i>	64
7.4	TD9 Autre application des fonctions <i>map</i> et <i>fold</i>	65
7.4.1	Soyons logiques	65
7.4.2	Affixes	65
7.5	TD9 Tri rapide par pivot	66
7.6	TD10 Définition des fonctions d'ordre sup. <i>map</i> et <i>fold</i>	66
7.7	TD10 Somme de pieces des monnaies	68

7.1 [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 puis réaliser une fonction `est_point_fixe` qui indique si un élément donné est un point fixe d'une fonction donnée.

Rappel 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 puis réaliser une fonction `est_idempotente` qui indique si une fonction donnée est idempotente sur un domaine de valeur donné (fourni par la liste des éléments du domaine).

Rappel Une fonction f est idempotente sur un domaine D si et seulement si $\forall x \in D : f(f(x)) = f(x)$.

7.2 **TD9** Composition, fonction locale

Mathématiquement, on note \circ la composition de fonctions. Pour toutes fonctions f et g telles que g soit définie sur l'ensemble des valeurs prises par f ,

$$(g \circ f)(x) \stackrel{\text{def}}{=} g(f(x))$$

- Q1. Spécifier puis implémenter la composition.
- Q2. Définir une fonction `incr` qui ajoute 1 à un entier. En déduire l'implémentation d'une fonction `plus_deux` qui ajoute 2 à un entier, en utilisant uniquement les fonctions `comp` et `incr`.
- Q3. Définir une fonction `fois_deux` qui multiplie un entier donné par 2. Puis, en utilisant les fonctions `comp`, `incr` et `fois_deux`, définir la fonction `f1` qui a chaque entier x associe $2 * x + 1$. Définir de manière similaire la fonction `f2` qui a chaque entier x associe l'entier $2 * (x + 1)$. Pour chacune de ces deux fonctions, proposer une variante utilisant `comp` et l'autre ne l'utilisant pas.

7.3 **TD9** Utilisation des fonctions d'ordre sup. *map* et *fold*

- Q1. Définir une fonction qui incrémente tous les éléments d'une liste d'entiers.

INDICATION Utiliser la fonction `map`

- Q2. Définir une fonction qui étant donnée une liste d'entiers retourne la liste des résultats de la division euclidienne de ces entiers par 3.
- Q3. Définir une fonction non récursive qui effectue la somme des entiers d'une séquence donnée.

INDICATION Utiliser la fonction `fold_left`

- Q4. Définir une fonction `maxs` qui retourne le maximum d'une séquence d'entiers **naturels** (\mathbb{N}) donnée.

INDICATION Utiliser la fonction `fold_left`

- Q5. Définir votre propre fonction `map` (`mymap`), de manière non récursive.
- Q6. Définir une fonction permettant d'inverser une séquence.

Remarque Cette fonction est prédéfinie par OCAML dans le module `List` sous le nom `rev`.

Q7. Définir une fonction permettant d'aplatir une séquence de séquence.

Remarque Cette fonction est prédéfinie par OCAML dans le module `List` sous le nom `flatten`.

Par exemple :

CODE CAML

```

flatten [ [1;2;3] ; [4] ; [5;6;7] ; [] ; [8;9] ; [] ] =
  [ 1;2;3;4;5;6;7;8;9 ]
;;
flatten [ [[1;2];[3;4]] ; [[5]] ; [[6;7];[8;9;10]] ] =
  [ [1;2];[3;4];[5];[6;7];[8;9;10] ] ;;
```

7.4 [TD9](#) Autre application des fonctions *map* et *fold*

7.4.1 Soyons logiques

Q1. Définir la fonction *conjonction* : appliquée à une séquence de booléens, *conjonction* retourne vrai si et seulement si tous les éléments de la séquence sont vrais.

Remarque Votre 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 mathématique universel (\forall). Ainsi, $(for_all\ p\ s) = vrai$ si et seulement si tous les éléments de s vérifient p .

Q2. En déduire une implémentation de la fonction *conjonction*.

Q3. Implémenter *for_all* en utilisant les fonctions `map` et `conjonction`.

Q4. Implémenter *for_all* uniquement à partir de la fonction `fold_left`.

Q5. Reprendre les questions précédentes en remplaçant *conjonction* par *disjonction* et *for_all* (\forall) par *exists* (\exists).

Q6. Définir la négation (\neg).

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

7.4.2 Affixes

Q8. Utiliser les fonctions `hd` et `List.rev` pour réaliser la fonction *dernier* qui retourne le dernier élément d'une séquence.

Q9. Utiliser la fonction `map` pour réaliser la fonction *ajouter_a_chaque* qui ajoute un élément en tête de chaque séquence d'une séquence de séquences.

ajouter_a_chaque 0 [[2;3];[5];[];[3;8]] = [[0;2;3];[0;5];[0],[0;3;8]]

Q10. Spécifier et implémenter la fonction *suffixes* :

$$\text{suffixes } [e_0; \dots; e_{n-1}] = [[]; [e_{n-1}]; [e_{n-2}; e_{n-1}]; \dots; [e_1; \dots; e_{n-1}]; [e_0; \dots; e_{n-1}]].$$

INDICATION Comparer les suffixes de $[e_0, e_1]$ et de $[e_0; e_1; e_2]$ et utiliser les fonctions *fold_left*, *hd* et *tl*

Q11. Spécifier et implémenter la fonction *prefixes* :

$$\text{prefixes } [e_0; \dots; e_{n-1}] = [[]; [e_0]; [e_0; e_1]; \dots; [e_0; \dots; e_{n-2}]; [e_0; \dots; e_{n-1}]].$$

INDICATION Utiliser *fold_right*.

Q12. Implémenter la fonction *prefixes* à partir des fonctions *suffixes*, *map*, et *rev*.

7.5 **TD9** Tri rapide par pivot

Algorithme du tri par pivot : Étant donnée une séquence *s* qui ne contient pas d'éléments répétés, le principe du tri par pivot consiste à :

1. choisir un élément de *s* qu'on appellera le pivot,
2. diviser la séquence *s* en deux sous-séquences *s1* et *s2* où :
 - *s1* est composée des éléments de *s* strictement inférieurs au pivot,
 - *s2* est composée des éléments de *s* strictement supérieurs au pivot,
3. trier *s1* et *s2* en appliquant récursivement la fonction *tri* à *s1* et à *s2*
4. construire la séquence triée en plaçant le pivot entre les deux séquences obtenues au 3 qui sont déjà triées.

Dans la suite, nous choisirons de prendre pour pivot le premier élément de la séquence.

Q1. Soit $s = [3; 5; 1; 0; 4; 2]$. Que valent le *pivot*, *s1* et *s2* ?

Q2. Réaliser la fonction *tri* en implémentant l'algorithme du tri par pivot.

INDICATION Utiliser une fonction d'ordre supérieur pour diviser la séquence *s* en deux sous-séquences *s1* et *s2*.

7.6 **TD10** Définition des fonctions d'ordre sup. *map* et *fold*

Nous allons définir les fonctions d'ordre supérieur *map*, *fold_left*, *find*, *filter* et *partition*, très utilisées en programmation fonctionnelle.

Ces fonctions sont déjà définies par OCAML dans le module `List`.

Les éléments d'une séquence non vide s de longueur n seront notés e_0, \dots, e_{n-1} . Autrement dit : $s = [e_0; \dots; e_{n-1}]$.

Afin de simplifier les notations, nous conviendrons que s est la séquence vide $[]$ si $n = 0$.

- Q1.** Spécifier et réaliser une fonction *map* qui applique une fonction f à tous les éléments d'une séquence s :

$$(\text{map } f [e_0; \dots; e_{n-1}]) = [f(e_0); \dots; f(e_{n-1})]$$

- Q2.** Spécifier et réaliser une fonction *fold.left* qui applique récursivement une fonction f à deux arguments sur un élément d'initialisation *init* et une séquence s , en parcourant s du premier au dernier élément. Autrement dit,

$$(\text{fold_left } f \text{ init } [e_0; \dots; e_{n-1}]) = (f \dots (f (f \text{ init } e_0) e_1) \dots e_{n-1})$$

Cette fonction reflète une méthode de découpage par la droite donnant une équation simple pour la séquence vide et une équation récursive qui indique comment évolue le résultat lorsqu'on ajoute un élément à droite à la séquence.

- Q3.** Considérons le cas classique d'un découpage à gauche et d'une fonction f décrite par deux équations dont seule la deuxième est récursive :

$$f [] = \text{init}$$

$$f pr :: suite = \text{feqrec } pr (f \text{ suite})$$

L'idée est de créer une fonction générique de ce type à deux paramètres : *init* et *feqrec*.

Spécifier et réaliser une fonction *fold.right* qui applique récursivement une fonction *feqrec* à deux arguments sur une séquence s et un élément d'initialisation *init*, en parcourant s du dernier au premier élément. Autrement dit,

$$(\text{fold_right } \text{feqrec } [e_0; \dots; e_{n-1}] \text{ init}) = (\text{feqrec } e_0 (\text{feqrec } e_1 (\dots (\text{feqrec } e_{n-1} \text{ init}) \dots)))$$

- Q4.** Considérons la suite $(u_n)_{n=0}^9$ formée des dix éléments $u_0 = u_2 = u_5 = u_9 = 0$ et $u_1 = u_3 = u_4 = u_6 = u_7 = u_8 = 1$. À l'aide de la fonction *fold.left*, calculer le dixième terme de la suite $(v_n)_{n=0}^{10}$ définie par $v_0 = 0$ et si n appartient à $\{0, \dots, 9\}$,

$$v_{n+1} = \begin{cases} v_n + 1 & \text{si } u_n = 0, \\ 2v_n & \text{si } u_n = 1. \end{cases}$$

- Q5.** Spécifier et réaliser une fonction *find* qui retourne dans une liste le premier élément d'une liste donnée satisfaisant un prédicat p donné.

- Q6.** Spécifier et réaliser une fonction *filter* qui, étant donnée une séquence, retourne les éléments qui satisfont un prédicat donné. L'ordre des éléments de la liste doit être conservé.

Q7. Spécifier et réaliser une fonction *partition* qui, étant donné un prédicat p et une séquence s , partitionne s en deux séquences s_1 et s_2 :

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

L'ordre des éléments de la liste doit être conservé.

Deux implémentations différentes sont réalisables selon que l'on utilise *filter* ou non.

7.7 **TD10** Somme de pièces des monnaies

Cet exercice a pour but d'utiliser des fonctions d'ordre supérieur pour énumérer tous les montants que l'on peut réaliser avec un exemplaire de chaque pièce de la monnaie européenne.

Ce problème s'est posé lors de la conception de la nouvelle monnaie afin de déterminer qu'elles étaient les pièces essentielles pour obtenir tous les montants (centime par centime) de 0 à 5 euros avec un minimum de pièces.

Pour résoudre ce problème on commence par définir un type *piece* qui représente le montant d'une pièce de monnaie et la séquence des pièces de la monnaie européenne.

- Q1.** Définir le type *piece* et la constante *les_pieces*.
- Q2.** Définir une fonction *ajouter_à_chaque* qui ajoute une pièce donnée dans une liste de listes de pièces. Autrement dit $ajouter_à_chaque(p, [s_1; s_2; \dots; s_n]) = [p :: s_1 ; p :: s_2; \dots; p :: s_n]$.
- Q3.** Rappeler à quoi sert la fonction *fold* et son principe de fonctionnement.
- Q4.** À l'aide des fonctions *fold* et *ajouter_à_chaque*, donnez une implantation de la fonction *les_combinaisons_possibles* qui prend en paramètre une séquence de pièces qui représente le contenu d'un porte-monnaie et qui construit la séquence de toutes les combinaisons possibles de ces pièces.
- Q5.** Définir de la fonction *tous_les_montants_possibles* qui prend en paramètre la séquence des pièces du porte-monnaie et retourne la séquence de tous les couples possibles de la forme (montant correspondant, séquences des pièces utilisées) et donner une réalisation de *tous_les_montants_possibles* à l'aide des fonctions *fold* et *les_combinaisons_possibles*.

Chapitre 8

Annales

Sommaire

8.1	Exam 11–12 : exercise, the omega function	69
8.1.1	Définition de <i>omega</i>	70
8.1.2	Échantillonnage d’une fonction	70
8.1.3	Extension d’ <i>omega</i> aux séquences	70
8.2	Exam 13–14 : exercise, the omega function, again!	71
8.2.1	Definition of Ω	71
8.2.2	Using the Ω function	72
8.2.3	Products of lists	72
8.3	Exam 13–14 : exercise, the filter function	73

8.1 Exam 2011–2012 (session 2) : exercice, la fonction omega

Considérons les spécifications des fonctions mathématiques Σ et Π suivantes :

$$\begin{array}{l|l} \mathbf{sigma} : (\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow \text{intervalle} \rightarrow \mathbb{Z} & \mathbf{pi} : (\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow \text{intervalle} \rightarrow \mathbb{Z} \\ (\text{sigma } f (b_i, b_s)) = f(b_i) + f(b_i + 1) + \dots + f(b_s) & (\text{pi } f (b_i, b_s)) = f(b_i) \times f(b_i + 1) \times \dots \times f(b_s) \\ = \sum_{k=b_i}^{b_s} f(k) & = \prod_{k=b_i}^{b_s} f(k) \end{array}$$

Généralisons l’opération effectuée par ces fonctions – une addition pour *sigma*, une multiplication pour *pi* – à une opération quelconque notée *op*, de type $\alpha \rightarrow \alpha \rightarrow \alpha$; nous obtenons la fonction *omega* :

$$\begin{aligned} \mathbf{omega} : (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\mathbb{Z} \rightarrow \alpha) \rightarrow \text{intervalle} \rightarrow \alpha \\ (\text{omega } op f (b_i, b_s)) &= f(b_i) op f(b_i + 1) op \dots op f(b_s) \\ &= \Omega_{k=b_i}^{b_s} f(k) \end{aligned}$$

NB : dans la sémantique de *omega*, l’opération *op* est notée de manière infix (comme le serait + ou \times). Si *op* est notée de manière préfixe, la sémantique de *omega* s’écrit :

$$(\text{omega } op f (b_i, b_s)) = (op \dots (op f(b_i) f(b_i + 1)) \dots f(b_s) \dots)$$

Les deux notations sont équivalentes.

- Q1.** (6pt) Implanter les fonctions *sigma* et *pi* en OCAML, **en utilisant la fonction *omega***. Vos implantations ne doivent pas être récursives.

8.1.1 Définition de *omega*

- Q2.** (8pt) Définir *omega* grâce à des équations récursives.
Q3. (6pt) Implanter *omega* en OCAML.

8.1.2 Échantillonnage d'une fonction

On souhaite maintenant obtenir les valeurs d'une fonction *f* sur un intervalle d'entiers, une manipulation parfois appelée *échantillonnage* :

$$(\text{echantillons } f \text{ } (b_i, b_s)) = [f(b_i) ; f(b_i + 1) ; \dots ; f(b_s)]$$

- Q4.** (4pt) Quel est le type et la valeur des expressions suivantes :
- a) (`echantillons (fun x -> x*x) (-2, 1)`)
 - b) (`echantillons (fun x -> sqrt(float_of_int x)) (0, 2)`)
- Q5.** (3pt) Donner le profil de la fonction *echantillons*
- Q6.** (6pt) Implanter la fonction *echantillons* en OCAML, de manière **non récursive**.

8.1.3 Extension d'*omega* aux séquences

L'objectif est maintenant de disposer d'une fonction *omega* sur les séquences plutôt que sur les intervalles comme c'est le cas dans les paragraphes précédents.

Sans changer de notation, on spécifie donc :

$$\begin{aligned} \mathbf{\omega} &: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \alpha) \rightarrow \beta \text{ list } (* \text{ non vide } *) \rightarrow \alpha \\ (\omega \text{ op } f [e_1 ; \dots ; e_n]) &= f(e_1) \text{ op } \dots \text{ op } f(e_n) \\ &= \Omega_{k=1}^n f(e_k) \end{aligned}$$

On notera que la fonction *omega* ainsi spécifiée manipule des séquences **non vides**, et que l'opération *op* est notée de manière infixé (comme le serait + ou ×).

- Q7.** (9pt) Quel est le type et la sémantique des expressions suivantes :
- a) `fun s -> (omega (+) (fun _ -> 1) s)`
 - b) `fun s -> (omega (*.) (fun e -> e) s)`
 - c) `fun p s -> (omega (||) p s)`

Rappel `||` est l'opérateur OCAML implantant la disjonction (le « ou »).

- Q8.** (8pt) Donner des équations récursives définissant *omega*.
- Q9.** (6pt) En déduire une implantation de *omega* en OCAML.
- Q10.** (4pt) Donner une autre implantation de *omega*, **non récursive**, grâce à la fonction *fold_left*.

Rappel Spécification de *fold_left* :

Soit *init* : α une valeur initiale et *co* : $\alpha \rightarrow \beta \rightarrow \alpha$ une fonction de combinaison,

fold_left (*co* : $\alpha \rightarrow \beta \rightarrow \alpha$) (*init* : α) : $\beta \text{ list} \rightarrow \alpha$

$(\text{fold_left } co \text{ init } [e_1 ; \dots ; e_n]) = (co \dots (co \text{ init } e_1) \dots e_n)$

8.2 Exam 2013–2014 (session 1) : exercice, encore la fonction omega!

La fonction *omega* – notée mathématiquement Ω – est une opération d’ordre supérieur sur les séquences non vides qui permet de réduire une séquence non vide à une valeur, à la manière des fonctions *fold* vues en cours/TD/TP :

8.2.1 Définition de Ω

SPÉCIFICATION

Profil $\Omega : (\alpha_2 \rightarrow \alpha_2 \rightarrow \alpha_2) \rightarrow (\alpha_1 \rightarrow \alpha_2) \rightarrow \text{seq } (\alpha_1)^* \rightarrow \alpha_2$

Sémantique : Soit *op* $\in \alpha_2 \rightarrow \alpha_2 \rightarrow \alpha_2$ une opération qu’on pourra noter de manière infixé, c’est-à-dire $(x \text{ op } y)$ au lieu de $(\text{op } x \ y)$.

$(\Omega \text{ op } f [e_1 ; \dots ; e_n]) = f(e_1) \text{ op } \dots \text{ op } f(e_n)$

RÉALISATION

(1) $(\Omega \text{ op } f [e]) = f(e)$

(2) $(\Omega \text{ op } f \text{ pr} :: \text{fin}) = f(e) \text{ op } (\Omega \text{ op } f \text{ fin})$, où *fin* $\neq []$

- Q1.** Implémenter Ω en OCAML.

8.2.2 Utilisation de Ω

Q2. Donner une implémentation non récursive utilisant Ω des fonctions suivantes :

- sommeCarres*, qui retourne la somme des carrés des éléments d'une séquence d'entiers non vide.
- nbPos*, qui retourne le nombre d'éléments positifs d'une séquence d'entiers non vide.
- nbCar*, qui retourne le nombre de caractères total d'une séquence non vide de chaînes de caractères.

INDICATION On pourra utiliser la fonction suivante :

Profil `String.length : string → ℕ`

Sémantique : `String.length(ch)` est le nombre de caractères de *ch*.

8.2.3 Produits de séquences

Produit de deux séquences

On définit le *produit* de deux séquences d'entiers de la façon suivante :

SPÉCIFICATION

Profil `prodSeq : seq(ℤ) → seq(ℤ) → seq(ℤ)`

Sémantique : $(\text{prodSeq } [a_1; \dots; a_n] [b_1; \dots; b_m]) = \begin{cases} [a_1 * b_1; \dots; a_n * b_n], & \text{si } n \leq m \\ [a_1 * b_1; \dots; a_m * b_m], & \text{si } n > m \end{cases}$

RÉALISATION

- $(\text{prodSeq } [] s) = s$
- $(\text{prodSeq } s []) = s$
- $(\text{prodSeq } pr_1 :: fin_1 pr_2 :: fin_2) = pr_1 * pr_2 :: (\text{prodSeq } fin_1 fin_2)$

Q3. Donner le résultat de l'évaluation des expressions suivantes :

- $(\text{prodSeq } [4; 2] [10; 20])$
- $(\text{prodSeq } [4; 2] [10; 20; 30])$
- $(\text{prodSeq } [2; 4; 3] [10; 20])$

Q4. Définir une mesure, et prouver que $\forall s_1, s_2 \in \text{seq}(\mathbb{Z})$, l'évaluation de $(\text{prodSeq } s_1 s_2)$ termine.

Produit de n séquences

On définit le *produit* de n séquences d'entiers ($n > 0$) de la façon suivante :

SPÉCIFICATION

Profil $prodNSeq : seq(seq(\mathbb{Z}))^* \rightarrow seq(\mathbb{Z})$

Sémantique : $(prodNSeq [s_1; \dots; s_n])$ est la séquence des produits des éléments de s_1, \dots, s_n .

Exemple : $(prodNSeq [[2;4;3] ; [10;20;30] ; [2;-1]]) = [40;-80;90]$

Q5. Donner une implémentation non récursive utilisant Ω de $prodNSeq$.

INDICATION On pourra utiliser la fonction $prodSeq$ du paragraphe précédent.

Applatissage d'une séquence

L'applatissage d'une séquence est spécifié comme suit :

SPÉCIFICATION

Profil $aPlat : seq(seq(\alpha))^* \rightarrow seq(\alpha)$

Sémantique : $aPlat([s_1; \dots; s_n])$ est la séquence des éléments de s_1, \dots, s_n .

Exemple : $(aPlat [[2;4;3] ; [10;20;30] ; [2;-1]]) = [2;4;3;10;20;30;2;-1]$

Q6. Donner une implémentation récursive directe de $aPlat$ (sans utiliser Ω).

Q7. En utilisant Ω , donner une implémentation non récursive de $aPlat$.

8.3 Exam 2013–2014 (session 2) : exercice, la fonction filter

La fonction $filter$ est une fonction (d'ordre supérieur) de la librairie OCAML *List* que l'on peut spécifier de la manière suivante :

SPÉCIFICATION

Profil $filter : (\alpha_1 \rightarrow \mathbb{B}) \rightarrow \alpha_1 list \rightarrow \alpha_1 list$

Sémantique : Pour un prédicat p et une liste $s = [e_1; e_2; \dots; e_n]$, $(filter\ p\ s)$ renvoie la liste $s' = [e'_1; e'_2; \dots; e'_k]$ telle que :

- les e'_i sont tous les éléments de s qui vérifient le prédicat p
- l'ordre des éléments de s est conservé dans s'

Exemples :

1. $(filter\ (fun\ x \rightarrow x > 10)\ []) = []$
2. $(filter\ (fun\ x \rightarrow x > 10)\ [1;14;3;18;25]) = [14;18;25]$

Q1. (3pt) Donner une implémentation (récursive) de la fonction $filter$.

On représente un ensemble de points du plan par la séquence de leurs coordonnées (x, y) dans un repère orthonormé d'origine O .

Q2. (1pt) En supposant que les coordonnées sont des réels, définissez le type $ensPoints$ représentant un tel ensemble de points.

On rappelle que la distance d'un point de coordonnées (x, y) à l'origine du repère est donnée par la formule suivante :

$$d = \sqrt{x^2 + y^2}$$

Dans la suite, on pourra utiliser la fonction OCAML `sqrt` qui donne la racine carrée d'un réel positif.

Q3. (2pt) En utilisant la fonction *filter* donner une implémentation de la fonction *dansDisque* qui prend en paramètre un ensemble de points e , un réel r , et qui renvoie l'ensemble des points de e situés dans le disque de centre O et de rayon r .

Q4. (2pt) Donner une implémentation **non récursive** de la fonction *filter* en utilisant une des fonctions d'ordre supérieur de la famille *fold* dont on rappelle les profils ci-dessous :

`fold_left` : $(\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta \text{ list} \rightarrow \alpha$

`fold_right` : $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \rightarrow \beta$

Quatrième partie

STRUCTURES ARBORESCENTES

Chapitre 9

Rappels de cours

Rappels

- *définir* = donner une définition,
définition = spécification + réalisation ;
- *spécifier* = donner une spécification (le « quoi »),
spécification = profil + sémantique + exemples et/ou propriétés ;
- *réaliser* = donner une réalisation (le « comment »),
réalisation = algorithme (langue naturelle) + implémentation (OCAML) ;
- *implémenter* = donner une implémentation (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

10.1	TD11	Des ensembles aux arbres	77
10.2	TD11	Nombre de nœuds et prof.	78
10.3	TD11	Profondeur d'un arbre	78
10.4	TD11	Présence d'un élément	78
10.5	TD12	Descendance	79
		10.5.1 Réalisation de <i>estDesc</i> à l'aide de fonctions intermédiaires	79
		10.5.2 Réalisation directe de <i>estDesc</i>	80
10.6	TD12	Propriétés des arbres	80
10.7	TD12	Niveau d'un nœud	81
10.8	TD12	Ascendants d'un nœud	81
10.9	TD12	Propriétés des nœuds d'un arbre	81
10.10	TD12	Dico sous forme arborescente	84

10.1 [TD11](#) Des ensembles aux arbres

On considère un ensemble S de 5 éléments quelconques $\{e_1, e_2, e_3, e_4, e_5\}$. On définit la *profondeur d'un arbre* comme le nombre d'éléments que contient sa branche la plus longue.

- 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 d'un arbre

- Q1.** Représenter graphiquement des arbres de profondeur 0, 1, 2 et 3 de deux manières :
- en dessinant l'arbre vide (graphiquement représenté par Δ),
 - sans dessiner l'arbre vide.

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	Δ	\vdots	\vdots	\vdots
b) sans l'arbre vide		\vdots	\vdots	\vdots

- Q2.** Compléter le tableau suivant :

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

- 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 (algorithme, terminaison, implanta-tion) d'une fonction polymorphe renvoyant la profondeur d'un arbre.

10.4 TD11 Présence d'un élément dans un arbre

- Q1.** Donner la définition (spécification, algorithme, terminaison) du prédicat poly-morphe $appArb$ qui indique si un élément appartient à un arbre d'éléments.
- Q2.** Donner une autre réalisation de $appArb$, non récursive, à l'aide de deux fonctions intermédiaires, que l'on prendra soin de spécifier.

INDICATION

- Vous savez tester la présence d'un élément dans une séquence.
- Saurez-vous transformer un arbre en séquence ?

La réalisation de ces deux fonctions n'est pas demandée.

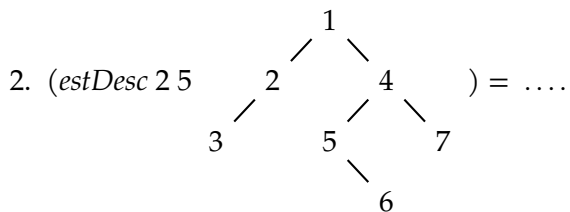
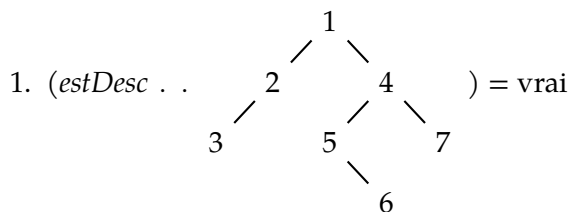
10.5 TD12 Descendance

Dans cet exercice, ainsi que dans ceux qui suivent, on pourra supposer que les éléments des arbres sont deux à deux distincts.

Les *descendants* d'un élément e dans un arbre a sont les nœuds du sous-arbre de a dont la racine est e .

Q1. Spécifier un prédicat *estDesc* qui indique si un nœud f est un descendant d'un nœud e dans l'arbre a . On posera par convention que si e et/ou $f \notin a$, $(estDesc\ e\ f\ a) = \text{faux}$

Exemples :



10.5.1 Réalisation de *estDesc* à l'aide de fonctions intermédiaires

La *descendance* est la séquence des descendants. Pour savoir si un nœud f est un descendant d'un nœud e dans a , il suffit de tester sa présence dans la descendance de e .

Q2. Spécifier une fonction appelée *descendance* donnant la descendance d'un élément dans un arbre.

Q3. En déduire une réalisation de *estDesc* ; on prendra soin de spécifier toute fonction intermédiaire nécessaire.

Algorithme : Utilisation de :

-
-

Pour réaliser la fonction *descendance*, on utilise une fonction appelée *sousArbre_deRac* qui, étant donné un élément e et un arbre a , donne le sous-arbre de a dont la racine est e . Si e n'est pas présent dans a , on considère que le sous-arbre de a de racine e est vide.

Q4. Définir *sousArbre_deRac*.

Q5. En déduire une implémentation de la fonction *descendance*.

INDICATION Utiliser, après avoir rappelé sa spécification, une fonction vue en cours donnant la séquence des nœuds d'un arbre selon un certain parcours de l'arbre.

10.5.2 Réalisation directe de *estDesc*

Q6. Donner une réalisation complète (algorithme, terminaison, implantation) de la fonction *estDesc* sans utiliser les fonctions intermédiaires des questions précédentes ; on prendra soin de spécifier toute autre fonction intermédiaire nécessaire.

10.6 **TD12** Propriétés des arbres binaires

Remarque Certaines des questions de ces exercices seront revisitées dans l'UE INF122B à l'aide du formalisme de preuve de la « déduction naturelle ».

Pour montrer qu'une propriété \mathcal{D} est vraie pour tout les $abin(\alpha)$ il suffit de montrer que la propriété est vraie lorsqu'on applique chaque constructeur qui permet d'obtenir un arbre binaire.

On doit donc montrer que :

1. \mathcal{D} est vraie pour le plus petit arbre que l'on peut construire : Δ .
Il faut donc montrer $\mathcal{D}(\Delta)$.
2. \mathcal{D} est vraie si on applique le constructeur $\text{Ab}(-, -, -)$ à deux sous-arbres g et d qui satisfont \mathcal{D} .
Il faut donc montrer $\forall (r, g, d) \in \alpha \times abin(\alpha)^2, (\mathcal{D}(g) \wedge \mathcal{D}(d)) \Rightarrow \mathcal{D}(\text{Ab}(g, r, d))$.

Exercice de rédaction de preuve

Étant donné un arbre binaire a , on désigne par :

- $n(a)$: le nombre de nœuds de a ,
- $b(a)$: le nombre de nœuds binaires vrais de a ,
- $v(a)$: le nombre d'arbres vides contenus dans a ,
- $f(a)$: le nombre de feuilles de a .

Q1. Rappeler ce qu'est un nœud binaire vrai, un nœud unaire et une feuille.

Q2. Formaliser puis prouver les propriétés suivantes par une *récurrence structurelle basée sur le type des arbres binaires*.

- a) Le nombre de nœuds binaires d'un arbre non vide a est égal au nombre de feuilles de a moins 1.
- b) Le nombre de sous-arbres vides d'un arbre a est égal au nombre de nœuds de a plus 1.

10.7 **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'au nœud e inclus. La racine est donc de niveau 1.

Propriété Soit e un nœud de niveau $n > 1$ dans un arbre $A_B(g, r, d)$ alors il se situe soit dans g , soit dans d . Le nœud e est de niveau $n - 1$ dans le sous-arbre (g ou d) auquel il appartient.

- Q1. Définir la fonction nbF_deNiv qui donne le nombre de feuilles à un niveau donné dans un arbre.
- Q2. Définir la fonction $nivElt$ 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.

10.8 **TD12** Ascendants d'un nœud

Définir une fonction $lesAsc$ qui donne la séquence au sens large des ascendants d'un nœud x dans un arbre a . « Au sens large » signifie que x est inclus dans ses ascendants. On admettra la propriété suivante :

Propriété $x \in a$ si et seulement si la séquence de ses ascendants n'est pas vide.

10.9 **TD12** Propriétés des nœuds d'un arbre

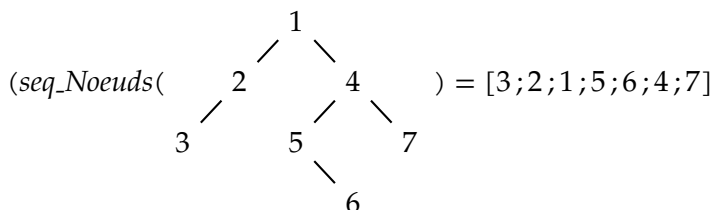
Considérons la fonction $seq_Nœuds$ qui construit la séquence des nœuds d'un arbre binaire parcouru selon l'ordre symétrique.

SPÉCIFICATION $seq_Nœuds$

Profil $seq_Noeuds : abin(\alpha) \rightarrow seq(\alpha)$

Sémantique : $seq_Noeuds(a)$ est la séquence des nœuds de a parcouru selon l'ordre symétrique.

Exemples :



L'exemple suivant illustre comment le calcul peut être effectué des feuilles vers la racine :

$$seq_Nœuds(\Delta) = []$$

$$\text{seq_Nœuds}(3) = [3] = [] @ (3 :: []) = \text{seq_Nœuds}(\Delta) @ (3 :: \text{seq_Nœuds}(\Delta))$$

$$\text{seq_Nœuds}(6) = [6] = [] @ (6 :: []) = \text{seq_Nœuds}(\Delta) @ (7 :: \text{seq_Nœuds}(\Delta))$$

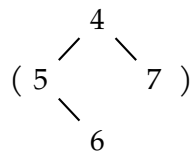
$$\text{seq_Nœuds}(7) = [7] = [] @ (7 :: []) = \text{seq_Nœuds}(\Delta) @ (7 :: \text{seq_Nœuds}(\Delta))$$

$$\text{seq_Nœuds}\left(\begin{array}{c} \\ \\ \\ \end{array}\right) = [3;2] = [3] @ (2 :: []) = \text{seq_Nœuds}(3) @ (2 :: \text{seq_Nœuds}(\Delta))$$

$$\text{seq_Nœuds}\left(\begin{array}{c} \\ \\ \\ \end{array}\right) = [5;6] = [] @ (5 :: [6]) = \text{seq_Nœuds}(\Delta) @ (5 :: \text{seq_Nœuds}(6))$$

$$\text{seq_Nœuds}\left(\begin{array}{c} \\ \\ \\ \end{array}\right) = [5;6;4;7] = \text{seq_Nœuds}\left(\begin{array}{c} \\ \\ \\ \end{array}\right) @ (4 :: \text{seq_Nœuds}(7))$$

$$\text{seq_Nœuds}\left(\begin{array}{c} \\ \\ \\ \end{array}\right) = \text{seq_Nœuds}\left(\begin{array}{c} \\ \\ \\ \end{array}\right) @ (1 :: \text{seq_Nœuds}\left(\begin{array}{c} \\ \\ \\ \end{array}\right))$$



$$\text{seq_Nœuds}\left(\begin{array}{c} \\ \\ \\ \end{array}\right) = [3;2] @ (1 :: [5;6;4;7]) = [3;2;1;5;6;4;7]$$

Seq_Nœuds est un cas particulier d'un schéma général plus de calcul, des feuilles vers la racine selon un parcours symétrique, d'un résultat de type β à partir d'un arbre d'éléments de type α , paramétré par :

1. $\text{valvide}(\alpha)$: le résultat à retourner pour un arbre vide (rappel : toute feuille est la racine d'un arbre binaire ayant un sous-arbre vide à gauche et à droite)

- opérateur $(\beta \rightarrow \alpha \rightarrow \beta)$: la fonction de calcul du résultat pour un arbre, à partir de sa racine et des résultats du calcul pour ses sous-arbres gauche et droit, appliquée répétitivement des feuilles jusqu'à la racine.

Pour `seq.Nœuds` :

- `valvide` : clairement la séquence vide (`[]`)
- opérateur : ajouter la racine à gauche de la séquence des nœuds du sous-arbre droit et concatèner le tout à droite de la séquence des nœuds du sous-arbre gauche.

On veut définir une fonction générique **abin.fold** (nommée ainsi par analogie avec les fonctions d'ordre supérieur sur les séquences) qui automatise ce type de calcul.

SPÉCIFICATION *abin.fold*

Profil *abin.fold* : $(\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow \text{abin}(\alpha) \rightarrow \beta$

Sémantique : *abin.fold* (opérateur, valvide, a) est le résultat de l'application répétée de opérateur des feuilles à la racine de a, valvide étant la valeur associée à un arbre vide.

A titre d'exemple, voici une fonction `seq_nœuds` non récursive utilisant la fonction `abin.fold` :

```
let seq_nœuds (a: 'elt abin) : 'elt list =
  let operation = fun resg r resd -> (resg @ (r :: resd)) in
  abin_fold operation [] a ;;

let abS (r: 'elt): 'elt abin = Ab(Av, r, Av) ;;
let abUNg (g, r: 'elt abin * 'elt): 'elt abin = Ab(g, r, Av) ;;
let abUNd (r, d: 'elt * 'elt abin): 'elt abin = Ab(Av, r, d) ;;

let ab7 : int abin = abS(7) ;;
let ab56: int abin = abUNd(5, (abS 6)) ;;
let ab4567: int abin = Ab(ab56, 4, ab7) ;;
let ab23: int abin = abUNg((abS 3), 2) ;;
let abtous: int abin = Ab(ab23, 1, ab4567) ;;

assert ((seq_nœuds ab4567) = [5;6;4;7]) ;;
assert ((seq_nœuds abtous) = [3;2;1;5;6;4;7]) ;;
```

Réaliser la fonction `abin.fold`.

Définir la fonction `sigma` qui calcule le nombre et la somme des nœuds d'un arbre d'entiers.

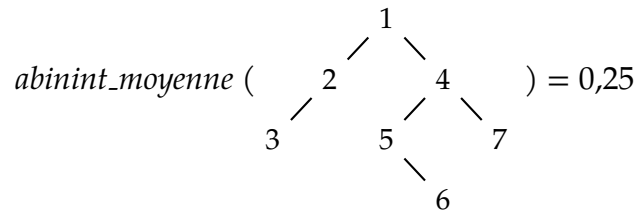
Définir une fonction **abinint_moyenne** qui calcule la moyenne des nœuds d'un arbre binaire d'entiers en utilisant une fonction auxiliaire **abinint_sigma** qui calcule le nombre et la somme des nœuds de l'arbre.

SPÉCIFICATION *abinint_moyenne*

Profil *abinint_moyenne* : $\text{abin}(\mathbb{Z}) \rightarrow \mathbb{R}$

Sémantique : *abinint_moyenne* (a) est la valeur moyenne des nœuds de a .

Exemples :

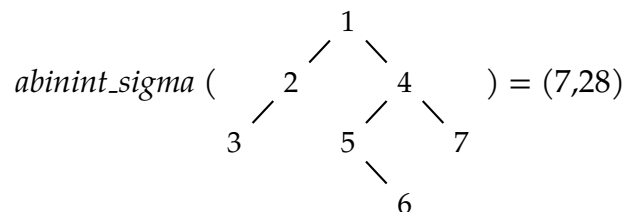


SPÉCIFICATION *abinint_sigma*

Profil *abinint_sigma* : $\text{abin}(\mathbb{Z}) \rightarrow \mathbb{Z} \times \text{Int}$

Sémantique : *abinint_sigma* (a) est le couple (nb, s) constitué du nombre nb et de la somme s des nœuds d'un arbre binaire d'entiers.

Exemples :



Donner le code caml d'une fonction **abinint_max** qui retourne la plus grandes des valeurs d'un arbre d'entiers **naturels**.

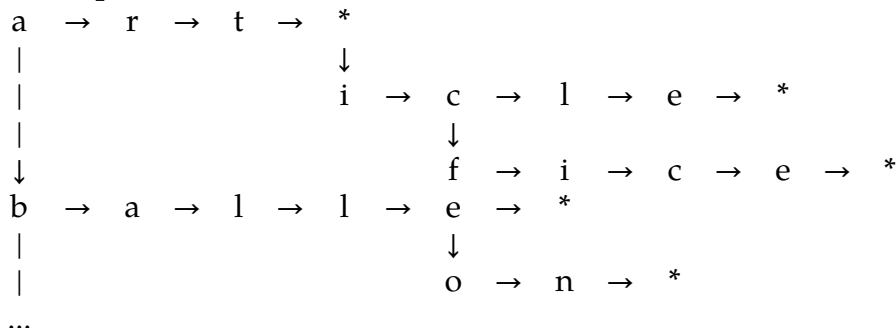
10.10 [TD12](#) Dictionnaire sous forme arborescente

On cherche à représenter sous forme compacte un dictionnaire¹ qui contient par exemple les mots suivants : art, article, artifice, balle, ballon, la, langage, langue. On pourrait représenter ce dictionnaire par la séquence de ses mots :

```
[ "art" ; "article" ; "artifice" ; "balle" ; "ballon" ; "la" ; "langage" ;
  "langue" ]
```

1. Cette terminologie n'a rien à voir avec les dictionnaires du langage Python.

On remarque que de nombreux mots ont un préfixe commun, c'est-à-dire un début identique. Par exemple, les mots « article » et « artifice » partagent le préfixe *arti* ; les mots « art » et « artifice » partagent le préfixe *art*. On utilise donc la représentation arborescente suivante, où les plus longs préfixes sont factorisés et où le caractère * sert à indiquer la fin d'un mot :



Cette représentation arborescente du dictionnaire a plusieurs avantages sur la représentation séquentielle ; notamment :

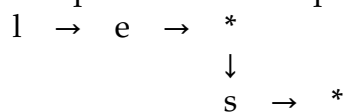
- la recherche d'un mot est plus rapide,
- l'empreinte mémoire de la structure de données est plus faible.

NB : dans cet exercice, et contrairement à ce que pourrait laisser croire les exemples ci-dessus, les mots ne sont pas ordonnés.

Représentation du dictionnaire

Q1. Compléter le dictionnaire avec les mots manquants. On prendra soin de dessiner un arbre binaire en représentant les arbres vides (Δ).

Q2. Définir un type appelé `dico` permettant de représenter un dictionnaire, puis donner la représentation du dictionnaire suivant :



Fonctions de manipulation d'un dictionnaire arborescent

Définir (spécification + réalisation) les fonctionnalités suivantes :

- Q3.** nombre de mots,
- Q4.** (*) présence d'un mot,

Q5. (*) nombre de mots d'une longueur donnée,

Q6. (**) séquence des mots,

Q7. (***) ajout d'un mot,

Q8. (***) suppression d'un mot.

Chapitre 11

Annales

Sommaire

11.1 Ex. exam 14–15 : Decision tree	87
11.2 Ex. exam 13–14 : all paths are leading to Rome	89
11.2.1 Terminology and definitions	89
11.2.2 The goal of the exercise	90
11.2.3 Tree of number of occurrences	91
11.3 Ex. exam 13–14 : Binary Search Trees	92
11.4 Pb exam 11–12 : n -ary trees	94
11.4.1 Un type pour les arbres n -aires	94
11.4.2 Nombre de feuilles	95
11.4.3 Appartenance	95

11.1 Exercice exam 2014–2015 : arbre de décision

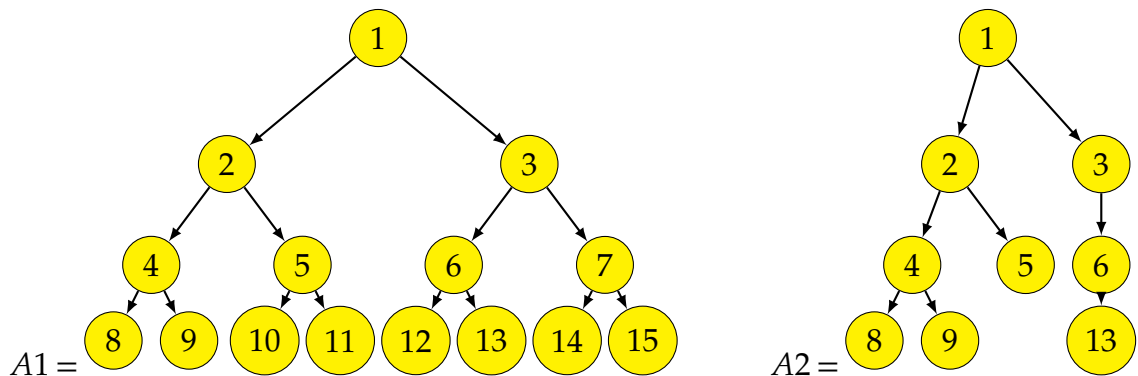
Dans cet exercice on appelle *arbre de décision* un arbre binaire **complet**. On rappelle qu'un arbre binaire est complet lorsque :

- sa **profondeur**¹ vaut p (avec $p \geq 0$)
- si $p > 0$ alors ses fils gauche et droit sont tous les deux complets et de profondeur $p - 1$.

On donne ci-dessous deux exemples d'arbres : A_1 est un arbre de décision, A_2 n'en

1. ou hauteur

est pas un.



On utilisera dans la suite le type `abin` pour représenter un arbre binaire dont les étiquettes sont de type quelconque (`'a`) :

```
type 'a abin =
  AV (* l'arbre vide *)
  | AB of 'a abin * 'a * 'a abin (* fils gauche, etiquette, fils droit *)
```

- Q1.** (2pt) Ecrire en Caml une fonction *profondeur* qui prend en paramètre un arbre binaire `a` et qui renvoie sa profondeur (c'est-à-dire la longueur du plus long chemin, en nombre de noeuds, entre la racine de `a` et une feuille de `a`).
- Q2.** (3pt) Ecrire en Caml une fonction *estArbDec* qui prend en paramètre un arbre binaire `a` et qui indique s'il s'agit ou non d'un arbre de décision.

Un arbre de décision `A` fournit une valeur en fonction d'une "séquence de décisions" exprimée par une séquence de booléens `L`. Cette valeur est obtenue en parcourant simultanément `A` et `L` selon l'algorithme suivant :

- si `L` est vide, la valeur retournée est l'étiquette associé à la racine de `A` ;
- si le premier élément de `L` est "vrai", alors il faut poursuivre le parcours sur le **fils gauche** de `A` et la **fin** de `L` ;
- si le premier élément de `L` est "faux", alors il faut poursuivre le parcours sur le **fils droit** de `A` et la **fin** de `L`.

Exemples :

- pour la liste de décision `[]` et l'arbre de décision `A1`, la valeur retournée est `1` ;
- pour la liste de décision `[true ; false ; true]` et l'arbre de décision `A1`, la valeur retournée est `10` ;
- pour la liste de décision `[false]` et l'arbre de décision `A1`, la valeur retournée est `3` ;

- pour la liste de décision `[false ; false]` et l'arbre de décision `A1`, la valeur retournée est 7.

Q3. (3pt) Ecrire en Caml une fonction *decision* qui prend en paramètre un arbre de décision `ad` et une séquence de décision `sd` et qui renvoie la valeur obtenue selon l'algorithme ci-dessus sous les hypothèses suivantes :

- `ad` est un arbre décision de profondeur `p` ;
- `|sd| < p` (avec `|sd|` le nombre d'éléments de `sd`).

Q4. (1pt) Expliquer pourquoi votre fonction *decision* termine (en indiquant notamment la *mesure* qu'il faut utiliser pour justifier sa terminaison).

11.2 Exercice exam 2013–2014 (session 1) : tous les chemins mènent à Rome

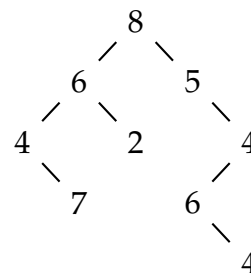
On considère des arbres binaires définis par le type suivant :

```
type 'a arbre =
  | V
  | N of 'a arbre * 'a * 'a arbre
```

Par exemple, l'arbre binaire suivant :

peut se dessiner ainsi :

```
let exArbre : int arbre =
  N (N (N (V, 4,
           N (V, 7, V)), 6,
         N (V, 2, V)), 8,
    N (V, 5,
      N (N (V, 6,
            N (V, 4, V)), 4, V)))
```



11.2.1 Terminologie et définitions

Ce paragraphe, explicatif, ne comporte aucune question.

Une *branche* est une séquence d'étiquettes de nœuds en partant de la racine et en allant vers les feuilles. Par exemple, l'arbre précédent comporte trois branches, qui sont :

- 1) 8, 6, 4, 7
- 2) 8, 6, 2

3) 8, 5, 4, 6, 4

On appelle *chemin* un préfixe d'une branche. Un préfixe est obtenu en retirant les n dernières étiquettes de la branche considérée ($n \geq 0$). Par exemple :

- 8, 6, 4
- 8, 6, 2
- 8, 5
- 8, 5, 4, 6

sont des chemins de `exArbre`.

11.2.2 But de l'exercice

Ce paragraphe, explicatif, ne comporte aucune question.

Le but de l'exercice est de déterminer les chemins qui mènent à une valeur donnée appelée *rome*. Par exemple, si *rome* vaut 4, on obtiendrait les chemins suivants :

- 8, 6, 4
- 8, 5, 4
- 8, 5, 4, 6, 4

Cependant, par pure malice, on veut éliminer les chemins qui seraient préfixe d'un autre. Ainsi dans l'exemple précédent, il ne resterait que :

- 8, 6, 4
- 8, 5, 4, 6, 4

À cet effet, la stratégie employée consistera à :

1. Construire un arbre auxiliaire obtenu en élaguant (supprimant), dans l'arbre d'origine, les sous-arbres qui ne contiennent aucune occurrence de *rome*.
2. Dans cet arbre auxiliaire, toutes les branches mènent à *rome*. Il suffira alors de construire la liste de toutes les branches de cet arbre.

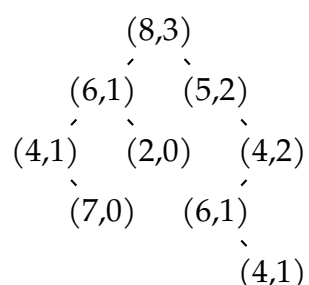
11.2.3 Arbre des nombres d'occurrences

Étant donné une valeur v et un arbre a quelconques, le *nombre d'occurrences* de v dans a est le nombre de fois où v apparaît dans a . Par exemple, le nombre d'occurrences de 4 dans `exArbre` est 3.

Q1. (1,5pt) Implémenter en OCAML une fonction `nbOcc` qui renvoie le nombre d'occurrences d'une valeur dans un arbre.

Étant donné une valeur v et un arbre a quelconques, on souhaite construire un arbre isomorphe à a , mais où les étiquettes des nœuds de a – appelons-les x – sont remplacées par un couple (x, n) où n est le nombre d'occurrences de v dans le sous-arbre de a qui possède x pour racine.

Par exemple, pour $v = 4$, l'arbre obtenu à partir de `exArbre` est :



Q2. (1,5pt) Implémenter en OCAML une fonction `arbNbocc` réalisant cette construction.

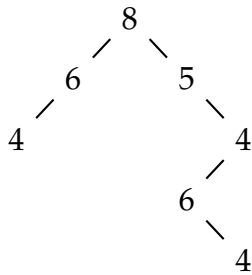
INDICATION Utiliser la fonction de la question précédente.

Étant donné un arbre de couples (x, n) tel qu'obtenu par la question précédente, on construit maintenant un arbre dans lequel :

- les sous-arbres où $n = 0$ sont *élagués* (remplacés par \vee),
- dans les nœuds restants, seul x est conservé.

Par exemple, à partir de l'arbre précédent, on obtient :

2. de même forme que



- Q3.** (1,5pt) Implémenter en OCAML une fonction *elaguer* réalisant cette opération.
- Q4.** (1pt) Dédurre des questions précédentes une fonction qui, étant donné un élément *rome* et un arbre *a* quelconques, construit l'arbre dont toutes les branches conduisent à *rome*, selon la stratégie expliquée en introduction.

On dit que l'arbre *b* est un *élagage* de l'arbre *a* si *b* est obtenu en remplaçant certains sous-arbres de *a* par \vee . Par exemple :

- $\text{elaguer}(a)$,
- \vee ,
- *a*,

sont tous des élagages de *a*, mais ce sont des cas particuliers.

- Q5.** (2,5pt) Étant donnés des arbres *a* et *b* quelconques, donner des équations récursives définissant la fonction *estElagage* qui détermine si *b* est un élagage de *a*.

11.3 Exercice exam 2013–2014 (session 2) : arbres de recherche

En informatique, un arbre binaire *de recherche* (ABR) est un arbre binaire *a* dans lequel chaque nœud est étiqueté par un entier *x* tel que :

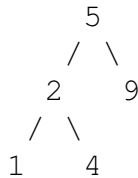
- chaque nœud du sous-arbre gauche de *a* est étiqueté par un entier inférieur ou égal à *x* ;
- chaque nœud du sous-arbre droit de *a* est étiqueté par un entier strictement supérieur à *x* ;
- les sous-arbres gauche et droit de *a* sont eux-mêmes des arbres binaires de recherche.

Binary Search Tree (ABR, in French) is a binary tree *a* where each node is labelled by an integer *x* such that :

- each node of the left sub-tree of *a* is labelled by an integer less or equal than *x* ;

- each node of the right sub-tree of a is labelled by an integer strictly greater than x ;
- the left and right sub-trees of a are binary search trees.

Par exemple :



Dans cet exercice, les ABR sont implémentés par le type suivant :

```

type abr =
  | v
  | N of abr * int * abr

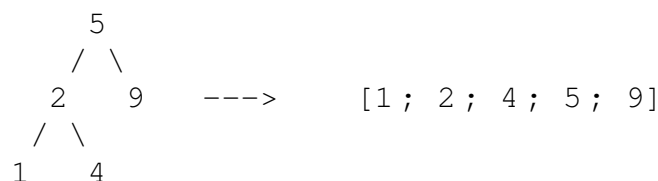
```

Pour insérer un entier i dans un ABR a , on l'ajoute comme *nouvelle feuille* de la manière suivante :

1. on recherche (récursivement) dans quel sous-arbre gauche ou droit de a l'entier i doit être inséré, jusqu'à atteindre une feuille F ;
2. on ajoute un nouveau nœud d'étiquette i comme fils gauche ou comme fils droit de F selon que i est supérieur ou inférieur à l'étiquette de F .

Q1. (2pt) Implémenter une fonction `insert` qui, étant donné un entier i et un ABR a , insère i dans a selon le principe décrit ci-dessus.
 Dans cette question, l'utilisation de toute fonction intermédiaire est interdite.

Q2. (2pt) Implémenter une fonction `parcours_s` qui, étant donné un arbre, retourne la séquence d'entiers obtenue en utilisant le parcours symétrique (sous-arbre gauche, racine, sous-arbre droit). Par exemple :



Pour trier une liste d'entiers, on utilise une fonction auxiliaire `tri_aux` qui, étant donné une liste l à trier et un ABR a , enlève le premier élément de la liste l et l'insère dans l'arbre à la bonne place, et cela tant qu'il y a encore des éléments dans l .

Q3. (2pt) Implémenter `tri_aux`.

Pour trier une liste d'entiers `l`, il suffit de démarrer avec la liste `l` et un arbre vide. On parcourt ensuite `l` en insérant tous les entiers dans le nouvel ABR qu'on construit à partir de l'arbre vide.

On utilise enfin la fonction `parcours_s` pour parcourir de manière ordonnée l'arbre obtenu.

Q4. (2pt) Implémenter la fonction `tri` en suivant l'algorithme décrit ci-dessus.

11.4 Problème exam 2011–2012 : arbres *n*-aires

On connaît les arbres 2-aires, plus communément appelés arbres binaires, dont une implantation en OCAML est :

```
type 'elt ab = V | N of 'elt ab * 'elt * 'elt ab
```

Q1. (6pt) Dans l'implantation ci-dessus :

- Qu'est-ce-que `'elt` ?
- Quel sont le type et la sémantique de `V` et `N` ?
- Donner un exemple d'une constante OCAML de type `'elt ab` comportant au moins deux nœuds et dessiner l'arbre correspondant.

On définit le *facteur de branchement* d'un arbre comme étant le nombre maximal de fils d'un nœud quelconque.

Q2. (2pt) Quel est le facteur de branchement des arbres binaires ?

L'objectif de ce problème est d'explorer une implantation des arbres *n*-aires, où le facteur de branchement *n* n'est pas fixé (en particulier à 2). En d'autres termes, le nombre de fils d'un nœud d'un arbre *n*-aire est un entier naturel quelconque.

11.4.1 Un type pour les arbres *n*-aires

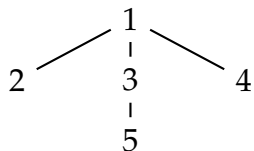
Dans ce problème, un arbre *n*-aire – appelé *ana* – sera implanté par un couple. Le premier élément du couple est la racine de l'arbre ; le deuxième élément est la séquence des sous-arbres. Le type des *ana* est donc :

```
type 'elt ana = 'elt * 'elt ana list
```

On constate que la notion d'arbre vide n'existe pas dans cette implantation.

Q3. (5pt)

- a) Donner un exemple, en OCAML, d'ana le plus petit possible. Combien a-t-il de nœuds ?
- b) Définir la constante OCAML `ex1` correspondant à l'arbre suivant :



Q4. (8pt) Implanter une fonction de conversion appelée *abVana* d'un arbre binaire non vide en ana de facteur de branchement 2 :

abVana : 'elt ab (* non vide *) -> 'elt ana
 (*abVana* x) est l'ana correspondant à l'arbre binaire x .

INDICATION Le filtrage doit comporter quatre cas.

11.4.2 Nombre de feuilles

- Q5.** (4pt) Spécifier une fonction *nbf* calculant le nombre de feuilles d'un ana. On prendra soin de donner le maximum de précision sur le co-domaine (l'ensemble d'arrivée) de *nbf*.
- Q6.** (8pt) Donner trois équations récursives définissant *nbf*.
- Q7.** (3pt) Détailler la trace de l'évaluation de *nbf*(1, [(2, [])]); préciser à chaque étape quelle équation a été appliquée.
- Q8.** (6pt) En déduire une implantation de *nbf* en OCAML.

11.4.3 Appartenance

La fonction *app*, qui teste l'appartenance d'un élément à un ana, est spécifiée comme suit :

app : 'elt -> 'elt ana -> bool

(*app* e a) est vrai si et seulement si e est un nœud de l'ana a .

Exemples :

1. (*app* 3 `ex1`) = vrai (la constante `ex1` a été définie au paragraphe 11.4.1)
2. (*app* 7 `ex1`) = faux
3. $\forall n, (\text{app } n \ (n, f)) = \text{vrai}$

Nous allons réaliser *app* de deux façons : à l'aide d'une fonction auxiliaire, puis directement.

Réalisation de *app* à l'aide d'une fonction auxiliaire

On appelle *forêt* une séquence d'arbres.

Considérons la fonction suivante :

```
appF : 'elt -> 'elt ana list -> bool
```

(*appF e f*) est vrai si et seulement si *e* est un nœud d'un des ana de la forêt *f*.

On peut implanter *app* à l'aide de *appF*, et réciproquement on peut implanter *appF* à l'aide de *app* ; on dit que *app* et *appF* sont *mutuellement* récursives.

Par exemple, pour savoir si 5 appartient à l'ana $\begin{array}{c} 1 \\ / \quad | \quad \backslash \\ 2 \quad 3 \quad 4 \\ \quad | \\ \quad 5 \end{array}$, puisque $5 \neq 1$,

on cherche à savoir si 5 est un élément de la forêt $[\bullet; \begin{array}{c} 2 \\ | \\ 3 \\ | \\ 4 \\ | \\ 5 \end{array}; \bullet]$. Pour cela, on teste

l'appartenance successive de 5 à l'arbre $\begin{array}{c} 2 \\ \bullet \end{array}$, puis à l'arbre $\begin{array}{c} 3 \\ | \\ 5 \end{array}$, etc.

Pour implanter des fonctions mutuellement récursives, on utilise la construction `let rec ... and`. Par exemple, les fonctions *pair* et *impair* dans \mathbb{Z} peuvent être implantées de cette façon :

```
let rec pair (x:int) : bool =
  x = 0 || if x = 1 then false else impair(x-1)
and impair (x:int) : bool =
  x != 0 && if x = 1 then true else pair(x-1)
```

Q9. (10pt) Donner une implantation mutuelle de *app* et de *appF*.

INDICATION S'inspirer du schéma d'implantation par récursivité mutuelle des fonctions *pair* et *impair* ci-dessus.

Réalisation directe de *app*

Q10. (6pt) Donner des équations récursives définissant *app* sans utiliser *appF*.