Programmation fonctionnelle et logique (INF6120)

Automne 2024 — Examen 1

3 heures

Samuele Giraudo

Université du Québec à Montréal

22 octobre 2024

Informations personnelles

Prénom : .....

Nom : .....

Code permanent : .....

Groupe : ☐ 20 (S. Giraudo)

Barème

Donné ici et dans les questions à titre indicatif uniquement.

1 : ___ /20 2 : ___ /20 3 : ___ /20 4 : ___ /20 5 : ___ /20

Instructions générales

1. Seules les notes manuscrites personnelles d'au plus huit pages au format A4 ou équivalent sont autorisées. Nom et prénoms doivent être écrits sur chacune de ces feuilles.
2. Tout appareil électronique est interdit (téléphones, montres, ordinateurs, calculatrices, *etc.*).
3. Il est interdit de dégrafer le sujet : aucun sujet avec des pages manquantes, détachées ou déchirées ne sera évalué.
4. Aucune sortie n'est permise durant l'examen.
5. La carte d'étudiant.e devra être présentée sur demande.
6. Les sacs et autres effets personnels non nécessaires à l'examen doivent être déposés sur le devant de la salle.
7. Les règles concernant le plagiat sont strictement appliquées.
8. Les questions sont majoritairement indépendantes et de difficulté non progressive.
9. Bien vérifier au début de l'épreuve que toutes les pages du sujet sont présentes (le nombre total de pages est spécifié en bas de cette page).
10. L'utilisation des fonctions et types des modules `Stdlib` et `List` est autorisée. Tout matériel provenant d'un autre module est interdit.

Ce document est composé de 14 pages.

1 Expressions

Question 1.1. *(3 points)* — Supposons avoir accès à un type `files` qui permet de représenter des files d'entiers, à un nom `file_vide` lié à la file vide, ainsi qu'à une fonction `enfiler` de type `int -> files -> files` telle que `enfiler i f` renvoie la file obtenue en enfilant l'entier `i` dans la file `f`. Écrire une expression dont la valeur est la file obtenue en enfilant successivement les entiers `2`, `8`, `1`, `-2` et `1`.

Question 1.2. *(3 points)* — Supposons avoir accès à une fonction `temps` de type `unit -> float` qui renvoie en secondes le temps passé depuis le 1er janvier 1970 avec une grande précision après la virgule. Écrire une expression dont la valeur est le couple de flottants dont la 1re coordonnée est la valeur renseignant cette durée et dont la 2e contient cette exacte même durée majorée d'une seconde.

Question 1.3. *(7 points)* — Supposons que `lst` est un nom lié à une liste d'entiers. Soit l'expression

```
if List.length lst = 0 then 1 else 2
```

Expliquer en quelle valeur cette expression s'évalue suivant la forme de `lst`. Expliquer ensuite le défaut principal de cette expression. Donner finalement une expression équivalente qui corrige ce défaut.

Question 1.4. *(7 points)* — Considérons l'expression

```
if test a b || test a c then
    if test a b then
        1
    else
        2
else
    3
```

où `test` est une fonction de type `int -> int -> bool` définie au préalable et `a`, `b` et `c` sont des noms liés chacun à un entier. Expliquer le défaut principal de cette expression. Donner ensuite une expression équivalente qui corrige ce défaut.

2 Fonctions

Question 2.1. *(3 points)* — Soit la fonction

```
let f =
    function x -> (function y -> (function x -> (function z -> (x, y, x, z))))
```

Donner l'affichage exact que produit l'interpréteur sur l'évaluation de la phrase

```
# f 1 2 3 4;;
```

Justifier la réponse.

Question 2.2. *(3 points)* — Soit la fonction

```
let f x y = (x, x + y, y)
```

Donner une définition alternative mais équivalente de `f` sans recourir au sucre syntaxique offert ni par les définitions via le `let` avec des paramètres, ni avec le `fun`. Dit autrement, il est obligatoire ici de n'utiliser que la construction de base `function` ainsi que le `let` sans paramètre.

Question 2.3. *(7 points)* — Soit la fonction

```
let rec compter lst x =
    match lst with
        |[] -> 0
        |x' :: lst' when x' = x -> 1 + compter lst' x
        |_ :: lst' -> compter lst' x
```

Observons l'appel suivant problématique

```
# compter (List.init 100000000 (fun _ -> 4)) 4;;
Stack overflow during evaluation (looping recursion?).
```

où la fonction `List.init`, de type `int -> (int -> 'a) -> 'a list`, est telle que `List.init n f` renvoie la liste de longueur `n` dont la valeur en position `i` est ce que renvoie la fonction `f` appliquée à `i`.

Expliquer pourquoi cette erreur a lieu et proposer ensuite une nouvelle version de la fonction `compter` qui pallie ce problème.

Question 2.4. *(7 points)* — Si n est un entier strictement positif, son *successeur* $s(n)$ est défini par

$$s(n) := \begin{cases} \dfrac{n}{2} & \text{si } n \text{ est pair,} \\ 3n+1 & \text{sinon.} \end{cases}$$

La *suite de Syracuse* de n est la suite d'entiers formée de n, de son successeur, du successeur de son successeur, *etc.*, jusqu'à obtenir 1. Par exemple, la suite de Syracuse de 10 est

$$10, 5, 16, 8, 4, 2, 1$$

et la suite de Syracuse de 14 est

$$14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.$$

La *longueur* d'une suite de Syracuse est le nombre d'entiers qu'elle contient. Par exemple, la longueur de la suite de Syracuse de 10 est 7 et celle de la suite de Syracuse de 14 est 18.

Écrire une fonction `longueur_syracuse` paramétrée par un entier `n` et qui renvoie la longueur de la suite de Syracuse de `n` lorsque `n` est strictement positif.

Voici quelques exemples :

```
# longueur_syracuse 1;;
- : int = 1
# longueur_syracuse 10;;
- : int = 7
# longueur_syracuse 14;;
- : int = 18
```

3 Inférence de types

Question 3.1. *(2 points)* — Définir une fonction `k` dont le type est `'a -> 'b -> 'a`. Justifier.

Question 3.2. *(4 points)* — Donner, en le justifiant soigneusement, le type de la fonction

```
let rec d x =
    d x
```

Question 3.3. *(7 points)* — Donner, en le justifiant soigneusement, le type de la fonction

```
let s f g x =
    f x (g x)
```

Question 3.4. *(7 points)* — Donner, en le justifiant soigneusement, le type de la fonction

```
let f x y =
    let g z =
        z mod 2 = 0
    in
    if x <= y + 1 then
        g
    else
        fun z -> not (g z)
```

4 Manipulation de types

Question 4.1. *(2 points)* — Considérons le type

```
type naturels_peano =
    |Zero
    |Succ of naturels_peano
```

Il permet de représenter les entiers naturels par le *codage de Peano*. Par exemple, sachant que le naturel 3 est le successeur du successeur du successeur du naturel 0, le nombre 3 est représenté par

```
Succ (Succ (Succ Zero))
```

Écrire une fonction `est_zero` paramétrée par un naturel de Peano `n` et qui teste si `n` est la représentation de 0.

Question 4.2. *(3 points)* — Cet exercice utilise le type `naturel_peano` défini dans la question 4.1.

Écrire une fonction `predecesseur` paramétrée par un naturel de Peano `n` et qui renvoie le prédécesseur de `n`. Lorsque `n` est le codage de 0, la fonction renvoie `n`.

Voici quelques exemples :

```
# predecesseur Zero;;
- : naturels_peano = Zero

# predecesseur (Succ Zero);;
- : naturels_peano = Zero

# predecesseur (Succ (Succ Zero));;
- : naturels_peano = Succ Zero
```

Question 4.3. *(7 points)* — Cet exercice utilise le type `naturel_peano` défini dans la question 4.1.

Écrire une fonction `addition` paramétrée par deux naturels de Peano `n1` et `n2` et qui renvoie le naturel de Peano qui est l'addition de `n1` et `n2`.

Voici quelques exemples :

```
# addition Zero Zero;;
- : naturels_peano = Zero

# addition (Succ Zero) Zero;;
- : naturels_peano = Succ Zero

# addition (Succ Zero) (Succ (Succ Zero));;
- : naturels_peano = Succ (Succ (Succ Zero))

# addition (Succ (Succ Zero)) (Succ (Succ Zero));;
- : naturels_peano = Succ (Succ (Succ (Succ Zero)))
```

Question 4.4. *(8 points)* — Soit le type

```
type 'a graphes = {
    sommets : 'a list;
    arcs : ('a * 'a) list
}
```

dont chaque valeur est un graphe orienté où le champ `sommets` est la liste de ses sommets et le champ `arcs` est la liste de ses arcs, représentés par des couples de sommets.

Écrire une fonction `ajouter_arc` paramétrée par un graphe `graphe` et un arc `arc`. La fonction renvoie le graphe obtenu en ajoutant l'arc `arc` au graphe `graphe`. Si l'arc est composé d'un sommet déjà existant dans `graphe`, il ne doit pas être ajouté au graphe renvoyé. Remarque similaire pour l'arc `arc` lui-même : s'il existe déjà dans `graphe`, il ne doit pas être ajouté au résultat. Donner aussi le type d'`ajouter_arc`.

5 Conception de types

Question 5.1. *(5 points)* — Expliquer la différence entre un type somme et un type produit. Donner un exemple concret et pertinent pour illustrer ces concepts.

Question 5.2. *(5 points)* — Définir un type `arbres` à deux paramètres `'noeuds` et `'feuilles` permettant de représenter tout arbre binaire dont chaque nœud interne porte une valeur de type `'noeuds` et chaque feuille porte une valeur de type `'feuilles`.

Question 5.3. *(5 points)* — Définir un type `cartes` qui permet de représenter des cartes à jouer. Chacune est constituée de trois informations : son élément (« eau », « terre », « feu », « air »), sa puissance (spécifiée par un entier) et par le fait d'être spéciale (spécifié par le fait que c'est le cas ou non). Il est possible de définir des types intermédiaires si besoin.

Question 5.4. *(5 points)* — Nous souhaitons définir un type `suites` qui permet de représenter des suites infinies d'entiers où sont connus

— le terme courant de la suite ;

— l'information pour calculer à partir d'un terme courant, le terme suivant de la suite.

Par exemple, la suite ayant 16 comme terme courant et dont le terme suivant se calcule en fonction de son terme courant x via la fonction $x \mapsto 2x$ spécifie la suite infinie des puissances de 2 à partir de 16, c'est-à-dire la suite

$$16, 32, 64, 128, 256, 512, 1024, 2048, \ldots$$

Pour illustrer ce qu'il est attendu de faire avec ce type, il doit être possible d'écrire la fonction `suivante` de type `suites -> suites` telle que `suivante s` renvoie la suite dont le terme courant est le terme suivant du terme courant de la suite `s`. Par exemple, cette fonction appliquée sur la suite précédente renvoie la suite

$$32, 64, 128, 256, 512, 1024, 2048, 4096, \ldots$$

Définir un tel type `suites`. Il n'est pas demandé ici d'écrire la fonction `suivante`.