Intra - ift2035 Nom:_____ Matricule:_____

Examen Intra

IFT-2035

October 28, 2020

Directives

- Répondre dans le *fichier de réponses* associé.
- Chaque question vaut 5 points, pour un total de 25 points.
- Les questions ne sont pas placées par ordre de difficulté.

0 Identification et code d'honneur

1. Écrire votre nom et matricule dans le fichier de réponses.
2. Promettre de faire cet examen sans faire recours à de l'aide extérieure.

1 Syntaxe

Soient les expressions suivantes en notation postfixe:

1. `a x y / d + -`
2. `x d - x - d -`
3. `a b + c ∧ d ∧`
4. `b a sin c * + d cos = not`
5. `a b < d c - d sqrt e * < not || f &&`
6. `x 0 = z a b + cons nil if`

Après s'être souvenu que la mise à la puissance (représentée par ∧ ci-dessus) est associative à droite, écrire ces expressions en format infixe, préfixe, et en format parenthèsé à la Slip/Lisp. En notation infixe utiliser un *minimum* de parenthèses.

2 Types

Le code ci-dessous utilise une version simplifiée de Haskell qui n'a pas de classes de types et où les nombres sont tous des entiers de type Int.

Donner le type des expressions ci-dessous.
Par exemple, la réponse pour $(2, 3)$ serait: (Int, Int)
Le type donné devrait être aussi polymorphe que possible.

1. $[(+), \lambda x \to \lambda y \to x]$
2. $\mathsf{let}\ f\ x = [x]\ \mathsf{in}\ (f, f\ 1)$
3. $([snd], [])$

Donner le type des trous $\bullet$ suivants.
Par exemple, la réponse pour $\bullet\ +\ 1$ serait: Int

4. $(\bullet\ 2\ 3) + 4$
5. $map\ (+)\ \bullet$
6. $\mathsf{let}\ f = \bullet\ \mathsf{in}\ (map\ snd\ f)\ ++\ (map\ fst\ f)$

Donner un exemple de code qui a le type demandé.
Par exemple, la réponse pour $\alpha \to \alpha$ serait: $\lambda x \to x$

7. $\mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int})$
8. $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$
9. $(\mathsf{Int}, \alpha \to [\beta])$
10. $((\mathsf{Int}, \mathsf{Int}) \to \alpha) \to \alpha$

Précisions: Les fonctions *map*, *fst*, et *snd* sont (pré)définies comme suit:

$map\ f\ [] = []$
$map\ f\ (x : xs) = f\ x : map\ f\ xs$

$fst\ (x, y) = x$
$snd\ (x, y) = y$

$++\ ::\ [\alpha] \to [\alpha] \to [\alpha]$

3 Portée

Soit le code ci-dessous qui est écrit en Haskell et utilise donc la portée statique:

```
λa◯-> λb◯->
   let c◯ b◯ = λd◯-> a◯ + b◯ in
   let d◯ = λa◯-> a◯ (a◯ b◯) in
   let a◯ (b◯, c◯) = b◯ (d◯, c◯)
   in λd◯-> c◯ (a◯, d◯)
```

Renommer toutes les variables (e.g. en y ajoutant un 0, 1, 2, ... dans le cercle) pour que chaque variable ait un nom différent des autres. Bien sûr ce renommage ne doit pas changer la sémantique du code.
Écrire juste les numéros ajoutés (dans l'ordre!) dans le fichier réponses.

4 Structures de données fonctionnelles

Soit le type suivant en Haskell qui définit un arbre binaire que l'on peut utiliser pour représenter une liste de valeurs, avec concaténation en temps constant:

```
data Rope a = Leaf Int [a] | Concat Int (Rope a) (Rope a)

ropeLength (Leaf   l _)   = l
ropeLength (Concat l _ _) = l

makeRope xs = Leaf (length xs) xs
concat r1 r2 = Concat (ropeLength r1 + ropeLength r2) r1 r2
```

1. Donner le type des fonctions ci-dessus.
2. Donner des type raisonnables pour les fonctions suivantes:
   (a) *ropeGet* : renvoie l'élément à une certaine position de la *rope*.
   (b) *ropeSplit* : sépare une *rope* en deux à une position donnée.
   (c) *ropeInsert* : insert une *rope* à une certaine position dans une autre.
3. Écrire le code de *ropeGet* et *ropeInsert*
4. Indiquer l'ordre de grandeur du nombre d'opérations nécessaires en temps CPU et en allocations mémoire pour exécuter *ropeGet* et *ropeInsert*.

5 Macros

Soit une macro `while` en ELisp, qui prend la forme suivante:

```
(while (<var> <cond>) <exp>)
```

Une telle expression devrait répéter `<exp>`, tant que `<cond>` est vrai. De plus la valeur de `<cond>` est accessible dans la variable `<var>` pendant l'évaluation de `<exp>`. Elle doit renvoyer comme valeur la dernière valeur renvoyée par `<exp>`. Vous avez reçu la version ci-dessous qui fonctionne plus ou moins mais souffre de quelques problèmes et devez les corriger:

```
(defmacro while (xc e)
  (let ((x (car xc))
        (c (cadr xc)))
    `(letrec ((loop (lambda (last)
                      (if (not ,c)
                          last
                        (let ((,x ,c))
                          (funcall loop ,e))))))
       (funcall loop nil))}))
```

Montrer le résultat de l'expansion de la macro pour les usages suivants:

1. `(while (cond (read-string)) (message "%s" cond))`

2. ```
   (while (while (is-good last))
     (setq last (get-last while)))
   ```

3. ```
   (while (c (car x))
     (let ((loop (cdr x)))
       (message "%s" c) (setq x loop)))
   ```

Certaines de ces expansions sont incorrectes. Indiquer lesquelles et pourquoi. Corriger la définition de la macro pour éliminer ces problèmes.