☐0 ☐0 ☐0 ☐0 ☐0 ☐0 ☐0 ☐0
☐1 ☐1 ☐1 ☐1 ☐1 ☐1 ☐1 ☐1
☐2 ☐2 ☐2 ☐2 ☐2 ☐2 ☐2 ☐2
☐3 ☐3 ☐3 ☐3 ☐3 ☐3 ☐3 ☐3
☐4 ☐4 ☐4 ☐4 ☐4 ☐4 ☐4 ☐4
☐5 ☐5 ☐5 ☐5 ☐5 ☐5 ☐5 ☐5
☐6 ☐6 ☐6 ☐6 ☐6 ☐6 ☐6 ☐6
☐7 ☐7 ☐7 ☐7 ☐7 ☐7 ☐7 ☐7
☐8 ☐8 ☐8 ☐8 ☐8 ☐8 ☐8 ☐8
☐9 ☐9 ☐9 ☐9 ☐9 ☐9 ☐9 ☐9

Principe de programmation
CC 2018

Nom, prénom :

.....

JEE

QCM noté sur 14 (remis sur 20). Il y a au moins une réponse juste et une fausse par question. Répondre toutes les réponses justes rapporte un point et répondre toutes les fausses rapporte -1 point, avec tout le panel entre les deux.[1]

Question [`conditionnel`] ♣ Pour remplacer un "if", je peux souvent utiliser

- ■ du pattern-maching
- ☐ une type-class
- ■ des gardes
- ☐ une récursion
- ☐ un sous-type

Question [`conditionnel2`] ♣ Le pattern-maching permet de

- ■ séparer des cas
- ☐ faire un appel récursif
- ■ accèder au contenu d'une structure
- ☐ typer correctement
- ☐ renvoyer des erreurs

Question [`fonction`] ♣ Une fonction est

- ☐ une classe
- ■ un programme
- ■ une valeur
- ☐ une structure de donnée
- ☐ Un `Ord`

Question [`fonction2`] ♣ L'argument d'une fonction peut être

- ☐ un type
- ■ une autre fonction
- ■ un entier
- ☐ elle même
- ■ un test d'égalité

Question [`typeint`] ♣ En Haskell, `Int`

- ■ implémente `Num`
- ■ implémente `Ord`
- ☐ est un type fonctionnel
- ☐ est un datatype
- ■ est un type

Question [`typeintlist`] ♣ En Haskell, `[Int]`

- ☐ implémente `Num`
- ■ implémente `Ord`
- ■ implémente `Enum`
- ☐ est un type fonctionnel
- ■ est un type

[1] S'il y a n réponse justes, une réponse juste vaut $\frac{1}{n}$ points et une réponse fausse $\frac{-1}{5-n}$ points.

Question [typeintfun] ♣ En Haskell, `Int->Int`

- ☐ implémente `Num`
- ☐ implémente `Ord`
- ☐ implémente `Enum`
- ■ est un type fonctionnel
- ■ est un type

Question [typesum] ♣ Lesquels de ces types sont correctes pour `sum x y = x+y`:

- ☐ `Int`
- ■ `Int -> Int -> Int`
- ■ `Num a => a -> a -> a`
- ☐ `Num`
- ☐ `Num -> Num -> Num`

Question [typeappend] ♣ Lesquels de ces types sont correctes pour `append x y = x++y`:

- ☐ `[a]`
- ☐ `[a] -> [b] -> [c]`
- ■ `[a] -> [a] -> [a]`
- ☐ `Num`
- ■ `[a->b] -> [a->b] -> [a->b]`

Question [typecons] ♣ Lesquels de ces types sont correctes pour `cons x y = x:y`:

- ☐ `[a]`
- ☐ `a -> a -> a`
- ■ `a -> [a] -> [a]`
- ☐ `List`
- ■ `Bool -> [Bool] -> [Bool]`

Question [typemap] ♣ Lesquels de ces types sont correctes pour `fmap`:

- ☐ `[a]`
- ☐ `a -> a -> a`
- ■ `(a->b) -> [a] -> [b]`
- ☐ `Functor`
- ■ `Functor f => (a->b) -> f a -> f b`

Question [commentaires] ♣ Les commentaires

- ☐ sont inutile
- ■ doivent être évités
- ☐ paraphrasent le code
- ■ pointent les optimisations
- ■ sont concis

Question [commentaires2] ♣ Les commentaires

- ☐ en mettre un maximum
- ☐ donnent le type
- ☐ marquent les cas impossibles
- ■ donnent la biblio
- ■ pointent les moments clés

Question [Erreurs] ♣ Dans un pattern-maching, un cas impossible

- ■ renvoi une exception
- ☐ ne doit pas être écrit
- ☐ renvoi une valeur bidon
- ☐ est mis commenté
- ☐ renvoi toujours le booléen faux

Question [Erreurs2] ♣ Si je trouve un cas impossible

- ■ je renvoi une exception
- ☐ je ne l'écrit pas
- ☐ je renvoi une valeur bidon
- ■ j'explique pourquoi en commentaire
- ☐ je renvoi le booléen faux

Question [Lecture] ♣ Pour aider le lecteur, je peux écrire

- ■ des commentaires
- ■ des types
- ■ des exception
- ■ des tests
- ☐ plusieurs codes (de la même fonction)

Question [Lecture2] ♣ Pour aider le lecteur, je peux ajouter

- ■ des noms de variables explicites
- ■ des indentations
- ■ des alias de types
- ☐ un benchmark
- ☐ des exemples en commentaire

Question [application] ♣ Avec le programme
`fun f = (\x -> f x) 3`

- [x] `fun` est équivalent à `(\f -> f 3)`
- [x] `fun` est équivalent à `($3)`
- [ ] on a une erreur de type
- [x] `fun (*2)` s'évalue en `6`
- [ ] `fun (*2) 4` s'évalue en `8`

Question [application2] ♣ Avec le programme
`fun f = (\x -> f ) 3`

- [ ] `fun` est équivalent à `(\f -> f 3)`
- [ ] `fun` est équivalent à `($3)`
- [ ] on a une erreur de type
- [ ] `fun (*2)` s'évalue en `6`
- [x] `fun (*2) 4` s'évalue en `8`

Question [application3] ♣ Avec le programme
`fun f = (\x -> f x x) 3`

- [x] `fun` est équivalent à `(\f -> f 3 3)`
- [x] `fun` est équivalent à `($3).($3)`
- [ ] on a une erreur de type
- [x] `fun (*)` s'évalue en `9`
- [ ] `fun (*2) 4` s'évalue en `8`

Question [commandes] ♣ Lesquels des programmes suivants sont-ils des commandes:

- [x] `type Func a = a -> a`
- [x] `fun x y = x+y`
- [x] `data Bool = True | False`
- [ ] `2*5`
- [ ] `Num a => a -> a`

Question [betaRed] ♣ La β-réduction

- [ ] est une opération sur les ARNs
- [x] permet de dire que `(\x->x)2` s'évalue en 2
- [ ] est une variante de l'α-réduction
- [x] est fondamental en prog. fonctionnelle
- [x] définit par `(\x->t) s` $\rightsquigarrow$ `t[s/x]`

Question [etaRed] ♣ La règle η

- [ ] est une opération sur les ARNs
- [ ] permet de dire que `(\x->x)2` s'évalue en 2
- [ ] est une variante de l'α-réduction
- [x] est une équivalence de programme
- [x] définit par `(\x->f x) s` $\simeq$ `f`

Question [show] ♣ En Haskell, `Show` :

- [x] est une type-class
- [x] peut être dérivée
- [x] implémenté par `[String]`
- [ ] implémenté par `(Int->Bool)`
- [ ] est une fonction

Question [Ord] ♣ En Haskell, `Ord` :

- [x] est une type-class
- [x] peut être dérivée
- [x] implémenté par `[String]`
- [ ] implémenté par `(Int->Bool)`
- [ ] est le type des listes ordonnées

Question [ARN] ♣ Dans un arbre rouge noir:

- [x] les fils de rouge sont noir
- [x] tous les chemins ont même hauteur de noirs
- [ ] tous les chemins ont même hauteur
- [ ] les fils de noir sont rouge
- [ ] on a un AVL

Question [ARNinsert] ♣ Lors de l'insertion dans un ARN, il faut:

- [x] fair remonter les paires père-fils rouges
- [ ] fair remonter les déficits en noeud noirs
- [ ] fair remonter les surplus de noeud noirs
- [x] potentiellement noircir la racine
- [ ] calculer la hauteur

Question [ARNsupr] ♣ Lors de la suppression dans un ARN, il faut:

- [ ] fair remonter les paires père-fils rouges
- [x] fair remonter les déficits en noeud noirs
- [ ] fair remonter les surplus de noeud noirs
- [ ] potentiellement noircir la racine
- [ ] calculer la hauteur

Question [`RandGen`] ♣ Un générateur aléatoir:

- [x] a besoin d'une seed
- [x] génère un élément aléatoire et une seed
- [ ] regarde l'heure
- [ ] va dans `devrandom`
- [x] est une fonction déterministe

Question [`datatypes`] ♣ Un data-type:

- [x] s'introduit avec le mot-clé `data`
- [x] permet le pattern-maching
- [x] dérive certaines type-classes
- [ ] dérive toutes les type-classes
- [ ] est un automate