☐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 [`fonction`] ♣ Une fonction est

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

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

- ☐ Int
- ■ une autre fonction
- ■ 3
- ☐ f
- ■ $(x == 3)$

Question [`fonction3`] ♣ Foncteur est

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

Question [`fonction4`] ♣ L'argument d'un foncteur f peut éventuellement être

- ■ Int
- ☐ un autre foncteur
- ☐ 3
- ☐ f
- ☐ $(x == 3)$

Question [`fonction5`] ♣ L'argument d'une fonction f peut éventuellement être

- ☐ Int
- ☐ un autre foncteur
- ■ 3
- ■ $(f3)$
- ☐ $(f == 3)$

Question [`typeint`] ♣ En Haskell, `Int` (avec potentiellement un wraper)

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

Question [`typeint2`] ♣ En Haskell, `Int` (avec potentiellement un wraper)

- ☐ implémente `Functor`
- ■ implémente `Monoid`
- ☐ est de type `List`
- ■ est le/un type de `minbound`
- ☐ est le/un type de `return`

[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 [typefloat] ♣ En Haskell, `Float` (avec potentiellement un wraper)

- [x] implémente `Num`
- [x] implémente `Ord`
- [ ] implémente `Enum`
- [ ] est un datatype
- [x] est un type

Question [typeintlist] ♣ En Haskell, `[Int]` (avec potentiellement un wraper)

- [ ] implémente `Num`
- [x] implémente `Ord`
- [ ] est de type `Maybe Int`
- [ ] est un type fonctionnel
- [x] est un type

Question [typeintlist2] ♣ En Haskell, `[Int]` (avec potentiellement un wraper)

- [ ] implémente `Functor`
- [x] implémente `Monoid`
- [ ] est de type `List`
- [x] est le/un type de `[minbound..]`
- [x] est le/un type de `return 3`

Question [typeintfun] ♣ En Haskell, `Int->Int` (avec potentiellement un wraper)

- [ ] implémente `Num`
- [ ] implémente `Ord`
- [ ] est de type `Maybe Int`
- [x] est un type fonctionnel
- [x] est un type

Question [typeintfun2] ♣ En Haskell, `[Int]->[Int]` (avec potentiellement un wraper)

- [ ] implémente `Functor`
- [x] implémente `Monoid`
- [ ] est de type `Fonction`
- [x] est le/un type de `fmap (+1)`
- [x] est un type fonctionnel

Question [typeNothing] ♣ En Haskell, `Nothing` (avec potentiellement un wraper)

- [ ] implémente `Num`
- [ ] implémente `Ord`
- [ ] est de type `Maybe Int`
- [ ] est un type fonctionnel
- [ ] est un type

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

- [ ] `Int`
- [x] `Int -> Int -> Int`
- [x] `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]`
- [x] `[a] -> [a] -> [a]`
- [ ] `Num`
- [x] `[a->b] -> [a->b] -> [a->b]`

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

- [ ] `[a]`
- [ ] `a -> a -> a`
- [x] `a -> [a] -> [a]`
- [ ] `List`
- [x] `Bool -> [Bool] -> [Bool]`

Question [commentaires] ♣ Les commentaires

- [ ] sont inutile
- [x] doivent être évités
- [ ] paraphrasent le code
- [x] pointent les optimisations
- [x] sont concis

Question [commentaires2] ♣ Les commentaires

- [ ] en mettre un maximum
- [ ] donnent le type
- [ ] marquent les cas impossibles
- [x] donnent la biblio
- [x] pointent les moments clés

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

- [x] 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

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

Question [Erreurs3] ♣ Pour indiquer une erreur je peux

- [x] utiliser un type `Maybe a`
- [x] renvoyer `undefined`
- [ ] renvoyer `Null`
- [x] renvoyer `Nothing`
- [ ] je renvoyer le booléen faux

Question [Erreurs3op] ♣ Pour indiquer une erreur, il ne faut pas

- [ ] utiliser un type `Maybe a`
- [ ] renvoyer `undefined`
- [x] renvoyer `Null`
- [ ] renvoyer `Nothing`
- [x] je renvoyer le booléen faux

Question [Lecture1] ♣ Pour aider/convaincre celui qui reprendra mon code, je peux ajouter

- [x] des commentaires
- [x] des types
- [x] des exception
- [x] un fichier de tests
- [ ] plusieurs codes de la même fonction

Question [Lecture1op] ♣ Pour aider/convaincre celui qui reprendra mon code, il ne faut pas

- [ ] des commentaires
- [ ] des types
- [ ] des exception
- [ ] un fichier de tests
- [x] plusieurs codes de la même fonction

Question [Lecture2] ♣ Pour aider/convaincre celui qui reprendra mon code, je peux ajouter

- [x] des noms de variables explicites
- [x] des indentations
- [x] des alias de types
- [x] un benchmark
- [ ] des exemples en commentaire

Question [Lecture2op] ♣ Pour aider/convaincre celui qui reprendra mon code, il ne faut pas

- [ ] des noms de variables explicites
- [ ] des indentations
- [ ] des alias de types
- [ ] un benchmark
- [x] des exemples en commentaire

Question [Lecture3] ♣ Pour aider/convaincre celui qui reprendra mon code, je peux ajouter

- [x] un README
- [x] une biblio
- [x] du polymorphisme
- [x] un fichier de benchmark
- [ ] des tests dans le fichier principal

Question [Lecture3op] ♣ Pour aider/convaincre celui qui reprendra mon code, il ne faut pas

- [ ] un README
- [ ] une biblio
- [ ] du polymorphisme
- [ ] un fichier de benchmark
- [x] des tests dans le fichier principal

Question [application] ♣ Avec le programme
`foo f = (\x -> f x) 10`

- [x] `foo` est équivalent à `(\g -> g 10)`
- [x] `foo` est équivalent à `($10)`
- [ ] on a une erreur de type
- [x] `foo (*2)` s'évalue en 20
- [ ] `foo (+2) 4` s'évalue en 6

Question [application2] ♣ Avec le programme
`foo f = (\x -> f ) 10`

- [ ] `foo` est équivalent à `(\g -> g 10)`
- [ ] `foo` est équivalent à `($10)`
- [ ] on a une erreur de type
- [ ] `foo (*2)` s'évalue en 20
- [x] `foo (+2) 4` s'évalue en 6

Question [application3] ♣ Avec le programme
`foo f = (\x -> f x x) 10`

- [x] `foo` est équivalent à `(\f -> f 10 10)`
- [x] `foo` est équivalent à `($10).($10)`
- [ ] on a une erreur de type
- [x] `foo (+)` s'évalue en 20
- [ ] `foo (+2) 4` s'évalue en 6

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 [Num] ♣ En Haskell, `Num` :

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

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

Question [`RandGen`] ♣ Un générateur aléatoire:

- [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

Question [`Maybe`] ♣ Maybe est

- [x] un type paramétré
- [x] un foncteur
- [x] une monade
- [ ] une type-class
- [ ] une fonction

Question [`Monoid`] ♣ Monoid est

- [ ] un type paramétré
- [ ] un foncteur
- [ ] une monade
- [x] une type-class
- [ ] une fonction

Question [`fmap`] ♣ fmap est

- [ ] un type paramétré
- [ ] un foncteur
- [ ] une monade
- [ ] une type-class
- [x] une fonction

Question [`foncteur`] ♣ un foncteur

- [x] est un type paramétré
- [ ] une fonction de fonction
- [ ] est une monade
- [x] définit fmap
- [ ] est une fonction

Question [`monade`] ♣ une monade

- [x] est un type paramétré
- [ ] définit mempty
- [x] est un foncteur
- [x] définit return
- [ ] est une fonction

Question [`monoid`] ♣ un monoid

- [ ] est un type paramétré
- [x] définit mempty
- [ ] est un foncteur
- [ ] définit return
- [ ] est une fonction

Question [`typeclasses`] ♣ Lesquels sont des type-class

- [ ] Maybe
- [x] Monad
- [x] Show
- [x] Foldable
- [ ] Just

Question [`typeclasses2`] ♣ Lesquels sont des type-classes d'ordre 2 (sur des types paramétrés)

- [ ] Maybe
- [x] Monad
- [ ] Show
- [x] Foldable
- [ ] Just

Question [`typeclasses1`] ♣ Lesquels sont des type-classes d'ordre 1 (sur des types non paramétrés)

- [ ] Maybe
- [ ] Monad
- [x] Show
- [ ] Foldable
- [ ] Just

Question [`constructeurs`] ♣ Lesquels sont des constructeurs

- [ ] Maybe
- [ ] Monad
- [ ] Show
- [ ] Foldable
- [x] Just

Question [fonctionsJust] ♣ Lesquels sont des fonctions

- ☐ Maybe
- ☐ Monad
- ☐ Show
- ☐ Foldable
- ■ Just

Question [typesMaybe] ♣ Lesquels sont des types

- ■ Maybe
- ☐ Monad
- ☐ Show
- ☐ Foldable
- ☐ Just

Question [Monades1] ♣ Lesquels sont des monades

- ■ `Maybe`
- ■ `[]`
- ■ `Either`
- ☐ `Set`
- ☐ `AutomateD`

Question [Monades1op] ♣ Lesquels ne sont pas des monades

- ☐ `Maybe`
- ☐ `[]`
- ☐ `Either`
- ■ `Set`
- ■ `AutomateD`

Question [Monades2] ♣ Lesquels sont des monades

- ■ `State Int`
- ■ `IO`
- ■ `Write Int`
- ☐ `Set`
- ☐ `ARN`

Question [Monades2op] ♣ Lesquels ne sont pas des monades

- ☐ `State Int`
- ☐ `IO`
- ☐ `Write Int`
- ■ `Set`
- ■ `ARN`

Question [Monades3] ♣ Lesquels sont des monades

- ■ `Maybe`
- ■ `State Int`
- ■ `IO`
- ☐ `ARN`
- ■ `Either`

Question [Monades3op] ♣ Lesquels ne sont pas des monades

- ☐ `Maybe`
- ☐ `State Int`
- ☐ `IO`
- ☐ `Either`
- ■ `ARN`

Question [MonadesUse1] ♣ Pour travailler avec des monade, j'utilise

- ■ le bind
- ■ le return
- ☐ un foncteur
- ☐ des gardes
- ☐ du pattern-matching

Question [MonadesUse2] ♣ Pour travailler avec des monade, j'utilise

- ■ la do-notation
- ■ `=<<`
- ☐ un Maybe
- ☐ `putStrln`
- ☐ des gardes

Question [MonadesUse3] ♣ Pour travailler avec des monades, j'utilise

- ☐ du pattern-maching
- ☐ des gardes
- ■ une do notation
- ☐ un monoid
- ☐ fmap

Question [conditionnel1] ♣ 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

- [x] séparer des cas
- [ ] faire un appel récursif
- [x] accéder au contenu d'une structure
- [ ] typer correctement
- [ ] renvoyer des erreurs

Question [conditionnel3] ♣ Les gardes permettent de

- [x] séparer des cas
- [ ] faire un appel récursif
- [ ] accéder au contenu d'une structure
- [ ] typer correctement
- [ ] renvoyer des erreurs

Question [conditionnel4] ♣ Pour remplacer un "if", je peux souvent utiliser

- [x] du pattern-maching
- [ ] un Maybe
- [x] des gardes
- [ ] une monade
- [ ] une do notation

Question [donotation] ♣ Une do-notation

- [x] permet d'écrire `bar <- foo`
- [x] permet de travailler dans l'environnement de la monade
- [x] a un style impératif
- [ ] fait un while
- [ ] est forcément dans un main

Question [egalite1] ♣ Lesquels de ces égalités sont toujours vrais

- [x] `(\x -> f x) ≃ f`
- [x] `(\x -> f (g x)) ≃ f . g`
- [x] `(\x -> x) 5 ≃ 5`
- [ ] `(\x -> x y) ≃ y`
- [ ] `(\x -> f (g x) ≃ (f g) x`

Question [egalite2] ♣ Lesquels de ces égalités sont toujours vrais

- [x] `(\x -> x y) ≃ $y`
- [x] `(\x -> x) 5 ≃ 5`
- [ ] `(\x y -> x (x y)) (+5) 2) 5 ≃ 13`
- [ ] `(\x -> y) ≃ 5`
- [ ] `(\x -> x y) ≃ y`

Question [egalite3] ♣ Lesquels de ces égalités sont toujours vrais

- [x] `(\x -> x+x) 5 ≃ 10`
- [x] `(\x -> f (g x)) ≃ f . g`
- [x] `(\x y -> x (x y)) (+5) 2) 5 ≃ (+10)`
- [ ] `(\x y -> x (x y)) (+5) 2) 5 ≃ 13`
- [ ] `(\x -> f (g x) ≃ (f g) x`

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

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

Question [typeMonad1] ♣ Lesquels sont des types correctes pour `foo f g x = f =<< (g =<< x)`

- [x] `(a->[b]) -> (b->[c]) -> (a->[c])`
- [x] `Monoid m => (a->m b) -> (b->m c) -> a-> m c`
- [x] `(a->Maybe a)->(a->Maybe a)->a->Maybe a`
- [ ] `a->[b] -> b->[c] -> a->[c]`
- [ ] `Functor m => (a->m b) -> (b->m c) -> (a->m c)`

Question [typeMonad1p] ♣ Lesquels sont des types correctes pour `foo f g x = f =<< (g =<< x)`

- [x] `(a->[b]) -> (b->[c]) -> (a->[c])`
- [x] `Monoid m => (a->m a) -> (a->m a) -> a-> m a`
- [ ] `(a->b->[c])->[a]->[b]->[c]`
- [ ] `a->[b] -> b->[c] -> a->[c]`
- [ ] `Monoid m => (a->b->m c)->m a->m b->m c`

Question [typeMonad2] ♣ Lesquels sont des types correctes pour `foo f x y =(\z-> f z =<<y)=<<x`

- [x] `(a->b->[c])->[a]->[b]->[c]`
- [x] `Monoid m => (a->b->m c)->m a->m b->m c`
- [ ] `(a->[b]) -> (b->[c]) -> (a->[c])`
- [ ] `Monoid m => m a`
- [ ] `func -> func`

Question [typeMonad2p] ♣ Lesquels sont des types correctes pour `foo f x y =(\z-> f z =<<y)=<<x`

- [x] `(a->b->[c])->[a]->[b]->[c]`
- [x] `(a -> b -> Maybe c) -> Maybe a->Maybe b->Maybe c`
- [x] `(a -> a -> State b c) -> State b a -> State b a -> State b c`
- [ ] `Monad`
- [ ] `Functor f`

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

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

Question [eqFoncteur1] ♣ Lesquels de ces équations doivent respecter les foncteurs:

- [x] `fmap id ≃ id`
- [x] `(fmap f).(fmap g) ≃ fmap (f.g)`
- [ ] `fmap f x ≃ (return.f) -<< x`
- [ ] `x ≃ return =<< x`
- [ ] `f =<< (g =<< x) ≃ (f.g) =<< x`

Question [eqMonad1] ♣ Lesquels de ces équations doivent respecter les monades:

- [ ] `fmap id ≃ id`
- [ ] `(fmap f).(fmap g) ≃ fmap (f.g)`
- [ ] `fmap f x ≃ (return.f) =<< x`
- [x] `x ≃ return =<< x`
- [ ] `f =<< (g =<< x) ≃ (f.g) =<< x`

Question [eqFoncteur2] ♣ Lesquels de ces équations doivent respecter les foncteurs:

- [x] `fmap (\x ->x) y ≃ y`
- [x] `fmap (\x -> f (g x)) y ≃ (fmap f) ((fmap g) y)`
- [ ] `fmap (fmap f) ≃ fmap f`
- [ ] `f =<< (g =<< x) ≃ (\ y -> f =<< (g y)) =<< x`
- [ ] `f (g =<< x) ≃ (f.g) =<< x)`

Question [eqMonade2] ♣ Lesquels de ces équations doivent respecter les monades:

- [ ] `fmap (\x ->x) y ≃ y`
- [ ] `fmap (\x -> f (g x)) y ≃ (fmap f) ((fmap g) y)`
- [ ] `fmap (fmap f) ≃ fmap f`
- [x] `f =<< (g =<< x) ≃ (\ y ->f =<< (g y)) =<< x`
- [ ] `f (g =<< x) ≃ (f.g) =<< x)`

Question [eqFoncteur3] ♣ Lesquels de ces équations doivent respecter les foncteurs:

- [x] `fmap (f.g) ≃ (fmap f).(fmap g)`
- [x] `(fmap f).(fmap g) ≃ fmap (f.g)`
- [ ] `fmap f x ≃ (return.f) =<< x`
- [ ] `f =<< (g =<< x) ≃ (\ y-> f =<< (g y)) =<< x`
- [ ] `f =<< return ≃ f`
- [ ] `f (g =<< x) ≃ (\y -> f =<< (g y)) =<< x`

Question [eqMonade3] ♣ Lesquels de ces équations doivent respecter les monades:

- [ ] `fmap (f.g) ≃ (fmap f).(fmap g)`
- [ ] `(fmap f).(fmap g) ≃ fmap (f.g)`
- [x] `fmap f x ≃ (return.f) =<< x`
- [x] `f =<< (g =<< x) ≃ (\y -> f =<< (g y)) =<< x`
- [ ] `f =<< return ≃ f`
- [ ] `f (g =<< x) ≃ (\y -> f =<< (g y)) =<< x`

Question [eqFoncteur4] ♣ Lesquels de ces équations doivent respecter les foncteurs:

- [x] `fmap id ≃ id`
- [x] `fmap (\x -> f (g x)) y ≃ (fmap f) ((fmap g) y)`
- [ ] `fmap f (fmap g) ≃ fmap (fmap (f.g))`
- [ ] `return =<< x ≃ x`
- [ ] `f =<< (return x) ≃ f x`

Question [eqMonade4] ♣ Lesquels de ces équations doivent respecter les monades:

- [ ] `fmap id ≃ id`
- [ ] `fmap (\x-> f (g x)) y ≃ (fmap f) ((fmap g) y)`
- [ ] `fmap f (fmap g) ≃ fmap (fmap (f.g))`
- [x] `return =<< x ≃ x`
- [x] `f =<< (return x) ≃ f x`