Principes de Programmation
TP6: Graphe et monade d'état

22 mars 2019

Exercice 1 (Graphes).
Un noeud du graphe est donné par une information et un liste de noeuds ; un graphe est alors une liste de noeuds :

```
type Nom = String
data Noeud  = NOEUD Nom [Noeud]
type Graphe = [Noeud]
```

Attention, le graphe devra rester bidirectionnel.

1. Copiez ce type et dérivez les classes habituelles.
2. Créez un graphe avec deux noeuds "noeud1" et "noeud2", pointant l'un sur l'autre.
3. Affichez ce graphe (en utilisant show).
4. Implémentez `show` correctement pour éviter ce soucis...
5. Écrivez une fonction :

   ```
   nouveauNoeudLie :: Graphe -> Nom -> Nom -> Graphe
   ```

   qui rajoute un noeud (1er nom donné) lié uniquement au noeud dont le nom est donné en second du graphe (celui en tête de liste), et ajoutez le lien inverse.
6. Testez cette fonction sur votre graphe pour ajouter "noeud0". Tout va bien ?
7. Écrivez une fonction

   ```
   getAdj2 :: Noeud -> [Noeud]
   ```

   donnant les noeuds à distance 2 de l'entrée.
8. Maintenant afficher les noeuds à distance 2 de "noeud2" ; que se passe-t-il ?
9. Comment remédier à ce problème ? Quelle est la complexité de la solution ?

En fait, le problème réel est le suivant : les données de haskell ne sont pas "mutable", c'est à dire que si l'on change quelque chose, on crée une nouvelle donnée et tous les pointeurs vers l'ancienne restent là où ils sont. Cela ajoute de la sécurité et ne coûte pas grand chose en terme de ressource, par contre il faut faire atention lorsque l'on a besoin de modifier une structure avec des boucles comme des automates ou des graphes.

Le meilleur moyen de s'en sortir est alors d'implémenter nous même des pointeurs ! Pour cela, c'est très simple, il faut changer le type du graphe par :

```
data Noeud = NOEUD (Set Nom)
type Graphe = Map Nom Noeud
```

L'information sert alors de clé pour retrouvé un noeud dans le graphe. Il n'y a qu'un seul pointeur réel sur un noeud : celui dans le graphe. On a juste créé un nouveau pointeur virtuel qui utilise l'information d'un noeud comme "adresse".

1. Importez `Data.Set` et `Data.Map` puis modifiez le type de graphe.

10. Écrivez une fonction d'ajout d'une information dans un graphe en donnant une clé fraîche :

    ```
    newEtatById :: Graphe -> Nom -> Graphe
    ```

11. Implémentez quelques fonctions utiles sur ce graphe et faites quelques tests...

Exercice 2 (Monade à état).
La monade à état `State` est déjà implémenté dans haskell. Elle a moralement la définition suivante :

```
type State s a = (s -> (a,s))
```

où `s` est le type de l'état courant et `a` est le type sur lequel on travaille.

En fait, il serait plus correcte de dire que c'est `(State s)` qui est une monade, ce pour tout type `s`. En fait, travailler dans `(State s)` veux dire que l'on dispose, dans son environnement, d'une "référence", ou d'un "pointeur" vers un objet de type `s`. Il est alors possible de lire et de modifier cet objet. C'est une autre façon de remplacer les "mutable" (voire exercice précédent).

Cette monade, en plus du `return` et du `(=<<)`, dispose de trois autres fonctions intéressantes : [1]

```
return        :: a -> State s a
return a      = (\s -> (a,s))

(=<<)         :: (a -> State s b) -> State s a -> State s b
f =<< u       = (\s -> let (a,s')= u s in f a s')
```

1. Atention, dans haskell, si vous utilisez les set et get de Control.Monad.State, ces fonctions ne sont pas définies par rapport à State, mais à n'importe quelle type générique de la classe MonadState, dont State est l'instanciation canonique.

```haskell
get           :: State s s
get           = (\s -> (s,s))

put           :: s -> State s ()
put s         = (\_ -> ((),s))

runState      :: State s a -> s -> (a,s)
runState u s = u s
```

`get` permet de récupérer l'état, `put` permet de changer l'état, et `runState` permet de récupérer le résultat étant donné un état de départ.

1. Importez `Control.Monad.State`.
2. En utilisant `set` et `get`, écrivez une fonction qui incrémente un état courant (de type `Int`) et l'affiche :

   ```haskell
   incr :: State Int Int
   ```

On considère des graphe définit par :

```haskell
data Noeud = NOEUD (Set Nom)
type Graphe = Map Nom Noeud
```

L'idée est que le graph est vu comme un mutable ; pour ca on suppose que le graphe fait partie de l'état, c'est à dire que l'on travail dans la monade *StateGraphe*.

3. Retranscrire la fonction `newEtatById` avec un type monadique :

   ```haskell
   newEtatById :: Nom -> State Graphe ()
   ```

Le soucis, maintenant est que l'on travail souvent avec deux monades à la fois car il faut toujours faire des recherche (due aux pointeurs) et que l'on a des Maybe qui apparaissent. Heureusement, Maybe et State intéragissent bien ensemble au sense où on peut créer une monade qui ai des état et qui puisse échouer.

On va travailler avec une monade `GraphEtat` qui est une combinaison de State et de Maybe. On crée cette monade avec la "formule magique"[2] suivante :

```haskell
type GraphEtat = StateT Graphe Maybe
```

4. écrire ce datatype et vérifiez que le résultat est bien une monade grace à la commande `:info Monad GraphEtat` dans ghci.

On a alors accès aux fonctions suivantes :

```haskell
lift        :: Maybe a -> GraphEtat a
```

qui agit comme le return, mais avec un mabe en entré

2. On utilise ici un "MonadTransformer" StateT qui compose deux monades en une seule

```
runStateT   :: GraphEtat a -> Graphe -> Maybe (a, Graphe)
```

qui permet de récupérer le résultat étant donné un état de départ si celui-ci n'a pas rendu d'erreur

```
get         :: GraphEtat Graph
```

qui retourne le graph courant

```
put         :: Graph -> GraphEtat ()
```

qui change le graphe courant

5. Voici un code, essayez-le et dites ce qu'il fait :

```
exist     :: Nom -> GraphEtat Bool
exist n = do
  graph <- get
  return (member n graph)
```

6. On peut échouer en renvoyant `lift Nothing :: GraphEtat a`, ou `fail` (qui est définit comme `fail = lift Nothing`). Écrire le programme de `newEtatById` pour qu'il échoue si l'état existe déjà :

```
newEtatById :: Nom -> GraphEtat ()
```

7. Écrire une fonction de recherche :

```
getById     :: Nom -> GraphEtat Noeud
```

qui échoue si elle ne touve pas le nom et renvoit le noeud présent dans le dictionaire sinon. On utilisera l'opérateur suivant de la librairie Data.Map :

```
(!?)        :: Ord k => Map k a -> k -> Maybe a
```

8. Écrire une fonction qui ajoute un lien :

```
newLienById :: (Nom,Nom) -> GraphEtat ()
```

elle doit échouer si les noms n'existent pas.

9. Utiliser une do-construction pour construire un petit graphe et afichez le à l'aide de la commande `runStateT mongraph empty` qui construit à partir du graphe vide.

1 À faire chez soi :

Exercice 3 (Graphe avancé).

1. Changez le type des graphe pour qu'il soit paramétrique :

```haskell
data Noeud a b = NOEUD a (Set (Nom,b))
type Graphe = Map Nom Noeud
type GraphEtat a b = StateT (Graphe a b) Maybe
```

Afin de pouvoir stoquer des information de type `a` dans les noeuds et des informations de type `b` dans les arrètes.

2. Mettez à jours les fonctions utilitaires.
3. Ajouter les changements de labels :

```haskell
setLabNoeud :: Nom -> a -> GraphEtat a b Noeud
setLabLien  :: Nom -> Nom -> b -> GraphEtat a b Noeud
```

4. Implémentez Kruskal

```haskell
kruskal :: GraphEtat a Int [(Nom,Nom)]
```

5. Implémentez Dijkstra

```haskell
dijkstra :: Nom -> GraphEtat IntEtendu Int [(Nom,Nom)]
data IntEtendu = Entier Int | Infinit
```

On supose qu'au début, tout est mis à Infinit.