# Principes de Programmation
# TP2: Implémentation d'Automates

17 janvier 2019

Le TP1 n'est pas nécessaire pour faire celui-ci. Il est tout de même préférable que vous soyez aller jusque la question 1.7 du TP1 ; au sens où l'on commence à peu près à ce moment en termes de difficulté.

*Exercice* 0 (Nouvelles définitions de types).
Voici deux définitions de types

```
data EtatD     = Noeud Bool (Char -> EtatD)
type AutomateD = EtatD
```

1. Chargez ces définitions de types dans *ghci*.
2. Dans *ghci*, tapez `:t Noeud`, que remarquez vous ? `Noeud` n'est pas un type, mais un constructeur, contrairement à `:Etat`. Contrairement au TP1, on utilise la même convention typographique pour les constructeurs (comme `Noeud`) et les types (comme `EtatD`) : des mots commençant par un majuscule. Il s'agit de la norme en Haskell. Attention à ne pas les confondre (utilisez la coloration syntaxique).
3. Dans *ghci*, tapez `:t Noeud Bool undefined` . Cela donne bien un état, comme prévu.
4. Maintenant tapes `:t Noeud Bool undefined :: AutomateD`, cela devient un Automate. Il semble qu'il s'agisse d'un cast, mais ca n'en est pas un : `AutomateD` et `EtatD` sont un seul et même type. Il s'agit d'un alias de type qui permet juste de rendre vos définitions plus claires. [1]

*Exercice* 1 (Automates déterministes).
Pourquoi a-t-on appelé cette structure un automate ?
L'idée, ici, est que l'état `(Noeud True f)` est un état final qui, en lisant un caractère `c`. De même, `Noeud False f` est un état non-final qui, par ailleurs, se comporte pareil.

Un Automate déterministe est la donnée d'un de ces état, qui sera considéré comme l'état initial.

1. Ici ce n'est pas très utile, mais ca le sera plus lorsque vous arriverez aux automates non-déterminites.

1. On se souvient (ou pas ?) qu'un automate finit traditionnellement définit comme un ensemble d'états, un état initial, des états finaux et une table de transitions qui associe à chaque état (dans lequel on est) et caractère (que l'on lit) un nouvel état (où on va).
   Essayez de vous convaincre que cette définition est équivalente à la définition traditionnelle sauf que l'on n'a pas accès à tous les états.. [2]

2. Quelle est l'automate (au sens traditionnel) correspondant au programme `monAutomate` suivant :

```
etat1 :: EtatD
etat1 = Noeud False (\x -> if (x=='a') then etat2 else etat1)
etat2 :: EtatD
etat2 = Noeud True  (\_ -> etat2)
monAutomate :: AutomateD
monAutomate = etat1
```

3. Quelle est l'automate (au sens traditionnel) correspondant au programme `monAutomate2` suivant :

```
transitions1 :: (Char -> EtatD)
transitions1 'a' = etat2
transitions1 'b' = etat3
transitions1  _  = etat1
etat1 = Noeud False transitions1
etat2 = Noeud True  (\x -> if (x=='b') then etat3 else etat2)
etat3 = Noeud False (\_ -> etat3)
monAutomate2 = etat1
```

4. Définissez un `monAutomate3 :: AutomateD` qui reconnaît les mots avec un nombre pair de `'a'`.

*Exercice* 2 (Reconnaissance de mot).

1. (dificile) Implémentez la fonction :

```
reconnait :: AutomateD -> String -> Bool
```

   telle que `(reconnait aut w)` vérifie si l'automate `aut` reconnaît le mot `w`. [3] Si vous avez du mal, jetez un oeil à l'indices 1.

2. Faites des tests en utilisant un `main` et une construction `do`. On verra cette construction en détail plus tard. Pour l'instant considérez qu'une construction `do` permet d'écrire, en style impératif, une série d'impressions sur sortie standard de la forme :

```
putStrLn "Hello World"
```

2. Pour l'instant on considère que l'ensemble des états sont quelque part en mémoire. On verra dans l'Exercice 4 qu'en fait on peut faire un automate avec un nombre infini d'états avec notre définition.
3. On rappellera qu'en Haskell, un String est la même chose qu'un tableau de caractères.

et d'instanciation de `String`s venant de l'entrée standard :

```
mot <- getLine
```

comme par exemple :

```
main = do
  putStrLn "Quel phrase analyser?"
  mot <- getLine
  if (reconnait monAutomate mot)
    then do putStrLn ("Cette phrase contient le caractere 'a'")
    else do putStrLn ("Cette phrase ne contient pas le caractere 'a'")
```

A noter que vous pouvez compiler un fichier `.hs` avec la commande `ghc` pourvu que le fichier comporte un `main`. Vous pouvez aussi faire vos tests dans ghci, bien entendu.

3. (dificile) Très souvent, un automate permet de vérifier la présence ou l'absence de sous-chaînes de caractères. Dans ces cas, on voit apparaître des états "puits" dont on ne peut pas sortir car on "sais déjà" que l'on va accepter ou refuser. Décrire le type algébrique (ADT) correspondant.

   Pour optimiser la reconnaissance dans ces cas, on voudrait accepter/refuser le mot dès que l'on atteint un état puits (plutôt que de boucler dessus jusque la fin du mot analysé). Modifiez la reconnaissence avec cette optimisation.

   En cas de blocage, vous pouvez consulter l'Indice 2 à la fin du TP.

   Pour la suite du TP, il n'est pas demandé de garder cet optimisation (qui complique l'exercice).

*Exercice* 3 (Automates non déterministes).
Un automate non déterministe sans "epsilon transitions" est de type :

```
data EtatND     = NoeudND Bool (Char -> [EtatND])
type AutomateND = [EtatND]
```

1. Essayez de vous convaincre que cette définition est équivalente à la définition traditionnelle.
2. Quelle est l'automate (au sens traditionnel) correspondant au programme `monAutomateND` suivant :

   ```
   etatND1 = Noeud False (\x -> if (x=='a') then [etatND1,etatND2] else [etatND1])
   etatND2 = Noeud False (\_ -> [etatND3])
   etatND3 = Noeud True  (\_ -> [])
   monAutomateND = [etatND1,etatND3]
   ```

3. Définissez un `monAutomateND2 :: AutomateND` qui reconnait les mots contenant la sous-chaîne `'.hs'`.
4. (Bonus) Trouvez un moyen d'ajouter les $\epsilon$-transitions.

Nous voulons maintenant déterminiser un automate non-déterministe. Pour cela, nous procédons récursivement sur chaque état. Nous n'avons pas besoin de vérifier si un état a déjà été exploré car l'on est en évaluation paresseuse : les nouveaux états seront créés à la volée pendant l'évaluation du mot. [4]

Rappelons que l'algorithme standard de déterminisation consiste à considérer un ensemble (ici une liste) d'état de l'automate non-déterministe de départ comme un seul état de l'automate déterministe d'arriver.

5 (Difficile) Écrivez une fonction

```
estTerminalList :: [EtatND] -> Bool
```

vérifiant que la liste d'états de l'automate non-déterministe est terminal en tant qu'état de l'automate déterministe résultant (càd que l'un des états de la liste est terminal). Vous êtes invités à utiliser la fonction `foldl` :

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

6 (Difficile) Écrivez la fonction de regroupement :

```
regroupe :: [Char -> [EtatND]] -> Char -> [EtatND]
```

Vous êtes invités à utiliser la fonction `concat` de la liste :

```
concat :: [[a]] -> [a]
concat = foldl (++) []
```

7 (Difficile+) Écrivez la fonction de déterminisation :

```
determinise :: AutomateND -> AutomateD
```

8 (Difficile) Faites des tests. Vous pouvez essayer de lancer des évaluations de fichier complets. Pour lire un fichier, placez les deux lignes suivantes sous une `do` construction :

```
monFichier <- openFile "monFichier.txt" ReadMode
contenu    <- hGetContents monFichier
```

La première ligne ouvre un fichier et la seconde le transforme en `String`.

4. Par contre on perd un peu en complexité. Pour palier ce problème, il faut utiliser de la mémoïsation, concept qui fera peut-être l'objet d'un autre TP.

# 1 À faire chez sois (bonus) :

*Exercice* 4 (Procrastiner c'est tricher).
On écrit le programme suivant :

```
aux 'a' = 1
aux 'b' = -1
aux  _  = 0

compteur :: Int -> EtatD
compteur n = Noeud (n == 0) (\c -> compteur (n + (aux c)))

compteurA :: AutomateD
compteurA = compteur 0
```

1. (Bonus) Que fait `compteurA` ? Est-ce un automate ?

La paresse de Haskell permet en fait de définir des structures infinies. En particulier, on peut ici tricher et définir un automate infini. [5]

2 (Bonus++) Définir un `AutomatD` reconnaissant les mots bien parenthésés.

3 (Bonus++) Sur le même modèle définir un programme `LireReg:: String -> Bool` qui reconnait une expression régulière écrite dans un `String`. Tout autre caractère que `(`, `)`, `|` et `*` est reconnu comme un caractère du langage.

4 (Bonus++) Réfléchir aux conséquences de ce pouvoir expressif.

*Exercice* 5 (Expressions rationelles (Bonus+++)).

1. Calculez la concaténation de deux automates .
2. Calculez l'union disjointe de deux automates.
3. Calculez l'étoile d'un automate.
4. Définissez un type `Reg` pour les expressions régulières.
5. Écrivez une fonction de création d'automate depuis une expression régulière :

   ```
   regToAutomate :: Reg -> AutomateD
   ```

*Indices* 1.
Sont indiqués ici les indices pour les questions difficiles.

1. Ici, on peut utiliser les types pour se guider.
   On dispose des arguments, mais aussi de la fonction elle-même :
   — d'un automate, qui n'est qu'un état `etat :: EtatD` ; c'est à dire :
   — d'un Booléen de terminaison `terminal :: Bool`
   — d'une fonction de transition `transitions :: Char -> EtatD`

5. au point que l'on est *Turing-complet*.

— d'un mot à lire `mot :: String`
— de la fonction elle même `reconnait :: AutomateD -> String -> Bool` que l'on peut appeler récursivement, mais sur des arguments plus petits (en l'occurrence le mot),

et on veut un Booléen.
Il faut lire le mot, cela veut dire qu'il faut récursser dessus. on a donc deux cas :

— soit le mot est vide, dans ce cas on ne peu pas faire grand chose d'autre que rendre le Booléen de terminaison, ca tombe bien car cela correspond à ce que l'on doit faire : sur un mot vide, on réussi si l'étai initial est aussi terminal,
— soit le mot est non-vide et il se décompose en un premier élément et la suite `x:suite`. Dans ce cas, `x` est un caractère et la seule chose que l'on peut faire avec est la donner à la fonction de transition, on a alors `(transitions x) :: EtatD`. Il nous reste alors à utiliser
  — la suite du mot `suite`,
  — le nouvel état `(transitions x) :: EtatD`,
  — de la fonction elle même `reconnait :: AutomateD -> String -> Bool` que l'on peut appeler récursivement, mais sur des arguments plus petits (par exemple la suite du mot...),

  on voit alors comment obtenir notre Booléen...

2. Il est beaucoup trop long de vérifier qu'une fonction de transition d'un état pointe sur lui même pour tout caractère lu. Il faut donc ruser et indiquer les états puits à la création, ce que l'on peut généralement faire facilement. Pour cela, on vous conseillera de changer le type des noeuds en :

```
data EtatD =   Noeud Bool (Char -> EtatD)
             | Puits  Bool
```

où les puits sont indiqués séparément des états et sans transitions (on garde le booléen pour savoir si le `Puits` est terminal ou non). Changer le reste du code en fonction.