S06 : Haskell (Types personnalisés)

Enseignant : Stéphane LE PEUTREC
Assistant : Jonathan Lauper

Instructions

- Deadline : jeudi suivant à 11:00

Exercice 1
Développez les fonctions suivantes (avec leur type).

- **product ' list** : cette fonction retourne le produit des éléments de la liste passée en paramètre. Développez une version de cette fonction avec un plis.

- **flatten' list** : cette fonction prend en paramètre une liste de listes et retourne en résultat la liste aplatie. Développez une version de cette fonction avec un plis.
  Exemple : flatten [[2,3], [4,2,7], [6,9]] retourne la liste [2,3,4,2,7,6,9]

- **deleteAll e list :** retourne la liste sans toutes ses occurrences de e. Développez une version de cette fonction avec un plis.

- **allDifferent l** : cette fonction retourne vrai si tous les éléments de la liste l sont différents. Développez quatre versions de cette fonction : une version récursive classique, une version avec accumulateur, une version avec programmation par continuation et une version avec un plis (fold).

Exercice 2
Dans cet exercice, on utilise le type de données suivant pour modéliser les arbres binaires ordonnés.

data BinaryTree a = EmptyTree | Node a (BinaryTree a) (BinaryTree a) deriving (Show, Eq, Ord)

Développez les fonctions suivantes (avec leur type).

- **insertInTree v tree** : cette fonction ajoute la valeur v dans l'arbre tree et retourne l'arbre obtenu.
  Exemple : insertInTree 4 (Node 3 EmptyTree EmptyTree) retourne l'arbre
  (Node 3 EmptyTree (Node 4 EmptyTree EmptyTree))

- **searchInTree v tree** : cette fonction retourne True s'il existe un nœud de valeur v dans l'arbre tree. Elle retourne faux sinon.
  Exemple : searchInTree 4 (Node 3 EmptyTree (Node 4 EmptyTree EmptyTree)) retourne True

- **sortTree tree** : cette fonction retourne la liste triée par ordre croissant des valeurs présentes dans l'arbre tree
  Exemple : sortTree (Node 3 (Node 2 EmptyTree EmptyTree) EmptyTree) retourne [2,3]

Programmation fonctionnelle

Exercice 3 : analyse lexicale - troisième étape

Dans cet exercice, on poursuit l'analyse lexicale commencée dans l'exercice 2 de la série 05. Les tokens reconnus pour ce langage sont :

- begin_block qui est exprimé par le caractère '{'
- end_block qui est exprimé par le caractère '}'
- begin_par qui est exprimé par le caractère '('
- end_par qui est exprimé par le caractère ')'
- seminolon qui est exprimé par le caractère ';'
- op_eg qui est exprimé par la suite de caractère "=="
- op_affect qui est exprimé par le caractère '='
- op_add qui est exprimé par le caractère '+'
- op_minus qui est exprimé par le caractère '-'
- op_mult qui est exprimé par le caractère '*'
- op_div qui est exprimé par le caractère '/'
- type_int qui est exprimé par la suite de caractères "int"
- cond qui est exprimé par la suite de caractères "if"
- loop qui est exprimé par la suite de caractères "while"
- value_int qui est exprimé par une suite de caractère respectant l'expression régulière $[0..9]^+$
- ident qui est exprimé par une suite de caractère respectant l'expression régulière $[a..zA..Z_][a..zA..Z0..9_]^*$

Les types introduits dans la série 05 sont en partie repris et transformés comme suit :

- type State = Int
- type Code = String
- type Transition = (State, (Char->Bool), State)
- data StateMachine = StateMachine State [State] [Transition]
- data Token = Token StateMachine Code

Exemples :

- le token begin_block est représenté par la valeur de type Token suivante :
  (Token (StateMachine 0 [1] [(0, (=='{'), 1)]) "begin_block")
  où (StateMachine 0 [1] [(0, (=='{'), 1)]) est l'automate qui reconnaît la chaîne de caractère "{"
  (0, (=='{'),1) est la seule transition de cet automate. Son état de départ est l'état 0, son état d'arrivée est l'état 1 et le caractère porté satisfait le prédicat (=='{')

La simplification adoptée dans la série 5 supposant que les tokens du texte à analyser sont séparés par des espaces est abandonnée. Les tokens du texte à reconnaître ne sont donc pas forcément séparés par des espaces. Exemple de texte : "int x; int y; x=y*2;"

Travail à faire

1) Représentez chacun des tokens de ce langage

Exemple : t1 = (Token (StateMachine 0 [1] [(0, (=='{'), 1)]) "begin_block")

Indications : vous pouvez utiliser des fonctions prédéfinies du module Date.Char telle que la fonction isDigit x qui retourne vrai si le caractère x est un chiffre et faux sinon

2) Implémentez les fonctions qui suivent :

- **recognizedFromState state text recognizedPart token** : cette fonction cherche à extraire du texte `text` un token en utilisant l'automate à état fini du token passé en paramètre. state est un état de l'automate de token, il représente l'état duquel on part pour analyser le texte. recognizedPart est un accumulateur contenant la partie du texte reconnue.

  Cette fonction retourne un triplet (code,rec,rest) où code est le code du token reconnu, rec est le dédut de text reconnu par le token et rest est la partie du texte qui suit la partie reconnue. Elle retourne le triplet ("","",text) si le début de text n'est pas reconnu par le token. Cette fonction est "gloutonne", elle reconnaît la plus grande partie possible de texte.

  Exemples :
  si t16 est la donnée de type Token permettant de reconnaître des identificateurs, l'appel `recognizedFromState 0 "toto=3;" "" t16` signifie que l'on cherche si le début du texte "toto=3;" est reconnu par le token t16 (si t16 est le token "ident").
  Cet appel retourne le triplet ("ident", "toto", "=3;") où "toto" est le plus grand identificateur au début du texte "toto=3;", "ident" est le code du token t16 et "=3;" est la partie du texte située après la partie reconnue (ici après "toto")

  l'appel `recognizedFromState 0 "{x=3;}" "" t16` signifie que l'on cherche si le début du texte "{x=3;}" est reconnu par le token t16.
  Cet appel retourne le triplet ("", "", "{x=3;}") car ce texte ne commence pas par un identificateur.

  Indications : Reprenez et adaptez au besoin les fonctions isFinalState et nextState de l'exercice 2 de la série 5.

- **getNextRecognizedToken text tokens** : cette fonction recherche quel est le token qui reconnaît le début du texte text et retourne en résultat le triplet (code,rec,rest) où code est le code du token reconnu, rec est le début de text reconnu par le token et rest est la partie du texte qui suit la partie du texte reconnu. Le paramètre tokens est la liste des tokens du langage.

  Exemple : l'appel `getNextRecognizedToken "toto=3;" lt` où lt est la liste des tokens du langage (lt=[t1,t2,...t16]) retourne le triplet ("ident", "toto", "=3;") car le texte commence par un identificateur, cet identificateur est "toto" et "=3;" est la partie du texte qui suit.

  Exemple : l'appel `getNextRecognizedToken "{x=3;}" lt` retourne le triplet ("begin_block", "{", "x=3;}") car le texte commence par le token {, et "x=3;}" est la partie du texte qui suit.

  Attention : `getNextRecognizedToken "while (x==3)" lt` doit retourner le triplet ("begin_loop", "while", " (x==3)") et non le triplet ("ident", "while", "(x==3)").

Indication : l'ordre dans lequel les tokens sont testés par cette fonction est significatif. Le token reconnu est celui qui reconnaît le plus long mot. Si deux tokens reconnaissent le même mot le gagnant est le premier qui a été testé.

- **lexAnalyse text tokens** : cette fonction retourne la liste des codes des tokens composant le texte text.
  Exemple : `lexAnalyse "toto=3;" lt` retourne la liste
  ["ident","op_affect","value_int","semicolon"]

  Indication : vous pouvez implémenter et utiliser la fonction trim ch qui retourne la chaîne de caractères ch sans ses premiers caractères qui sont des espaces. La fonction isSpace c du module Data.Char retourne True si c est un des caractères de séparation (espace, tabulation,etc)