S05 : Haskell (Récursivité terminale)

Enseignant : Stéphane LE PEUTREC
Assistant : Jonathan LAUPER

Instructions

- Deadline : jeudi suivant à 11:00

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

- **fibo n** : cette fonction retourne la valeur de fibonacci de n. Développez deux versions de cette fonction : une version avec accumulateur et une version avec programmation par continuation.

- **product ' list** : cette fonction retourne le produit des éléments de la liste passée en paramètre. Développez deux versions de cette fonction : une version avec accumulateur, une version avec programmation par continuation.

- **flatten' list** : cette fonction prend en paramètre une liste de listes et retourne en résultat la liste aplatie. Développez deux versions de cette fonction : une version avec accumulateur, une version avec programmation par continuation.
  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 deux versions de cette fonction : une version avec accumulateur, une version avec programmation par continuation.

- **insert' e list** : cette fonction prend en paramètre une valeur et une liste triée par ordre croissant et insère l'élément à sa place dans la liste. Développez deux versions de cette fonction : une version avec accumulateur et une version avec programmation par continuation.
  Exemple : insert 7 [2,5,8,9] retourne la liste [2,5,7,8,9]

Exercice 2 : analyse lexicale - deuxième étape

Dans cet exercice, on poursuit l'analyse lexicale commencée dans l'exercice 2 de la série 03. Le but de cet exercice est de transformer un texte source d'un mini langage en une suite de tokens. 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 03 sont en partie repris et complétés comme suit :

- un état d'un automate est modélisé par un entier
  type State = Int

- une transition d'un automate est modélisée par un triplet : l'état de départ de la transition, une fonction booléenne qui décrit le caractère porté par la transition et l'état d'arrivée de la transition
  type Transition = (State,(Char->Bool),State)

- un automate est modélisé par un triplet : l'état initial de l'automate, la liste de ses états finaux, et la liste de ses transitions
  type StateMachine = (State, [State], [Transition])

- le code représentant un token est modélisé par une chaîne de caractère
  type Code = String

- un token est modélisé par un couple : l'automate modélisant le token et son code
  type Token = (StateMachine,Code)

Exemples :

- le token begin_block est représenté par le doublet : ((0, [1], [(0, (=='{'), 1)]), "begin_block")
  où (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 (=='{')

- le token op_eg est représenté par le doublet ((0,[2],[(0,(=='='),1),(1,(=='='),2)]),"op_eg")
  où (0,[2],[(0,(=='='),1),(1,(=='='),2)]) est l'automate qui reconnaît la chaîne de caractère "=="

Dans cet exercice, par souci de simplification, on suppose que tous les tokens du texte à analyser sont séparés par des espaces. Exemple de texte : "int x ; int y ; x = y * 2 ; "

Travail à faire :

- Représentez chacun des tokens de ce langage
  Exemple : t1 = ((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

Programmation fonctionnelle

- Implémentez les fonctions qui suivent :

    - **getToken :: String -> [Token] -> Code** : cette fonction prend une chaîne de caractères et retourne le code du token correspondant à cette chaîne

      Exemple : l'appel `getToken "toto" <liste des tokens du langage>` retourne "ident" car "toto" est reconnu par l'automate du token "ident"

      Indications :
      cette fonction crée une exception si la chaîne de caractères passée en paramètre ne correspond à aucun token. Vous pouvez générer une exception grâce à la fonction `error ch`. Elle crée une exception et affiche la chaîne de caractères ch.

      Reprenez et adaptez au besoin les fonctions isToken, recognizedFromState, isFinalState, nextState de la seconde partie de l'exercice 2 de la série 3.

    - **lexAnalyse :: String -> [Code]** : cette fonction prend un texte source en paramètre et retourne la liste des tokens correspondant.

      Indication : Vous pouvez utiliser la fonction words qui permet de séparer une ligne de texte en plusieurs mots. Exemple : l'appel `words "int a ; int b ;"` retourne la liste ["int", "a", ";", "int", "b", ";"]

      Exemple : l'appel `lexAnalyse "int x ; int y ; x = y * 2 ; "` retourne la liste ["type_int", "ident", "semicolon", "type_int", "ident", "semicolon", "ident", "op_affect", "ident", "op_mult", "value_int", "semilocon"]