TP #4 : Codage de HUFFMAN

Organisation et cheminement du TP

Le TP s'organise en 3 parties principales (où l'on suppose l'arbre de `Huffman` déjà construit) :

1. Codage d'un texte
2. Décodage d'un texte
3. Expérimentations et calcul du taux de compression

La construction de l'arbre de `Huffman` est traitée Section #4 en fin d'énoncé sous la forme de questions dont les réponses sont dans le fichier `initTP4.hs`. Selon le temps qu'il vous restera après avoir traité les 3 principales parties, vous pourrez :

- soit regarder le code pour comprendre la construction de l'arbre
- soit répondre vous mêmes aux questions (sachant que vous avez un corrigé en cas de besoin)

Enfin, les diapos de la présentation faite au CM #7 sont accessibles en ligne sous `eCampus`.

Définition de types

On définit les types utilisateur `(Tree a)` et `Etape` ainsi le type synonyme `Chemin` :

```
data Tree a = Tip a | Bin (Tree a) (Tree a)
    deriving Show

data Etape = Gauche | Droite
     deriving Show

type Chemin = [Etape]
```

Soit `ht` l'arbre de `Huffman` construit à partir de la liste ordonnée des fréquences `fr` (fichier `initTP4.hs`).

```
fr = [('k',1), ('w',1), ... ('g',21), ('f',22), ('v',28), ... ('s',332), ('t',360), ('e',713)]

ht = (Bin (Bin (Tip 'e') (Bin (Bin (Bin (Bin (Bin (Tip 'x') ... (Bin (Tip 's') (Tip 't'))))))
```

1 Codage

Pour tester les fonctions, on utilisera l'arbre de `Huffman ht` donné comme exemple.

1. Définir la fonction `(appartient t c)` qui détermine si un caractère `c` appartient à un arbre de `Huffman t` :

   ```
   > appartient ht 'c'   ==> True
   > appartient ht '-'   ==> False
   ```

   Indications

   ```
   appartient :: Tree Char -> Char -> ?
   appartient (Tip y) x =
   appartient (Bin t1 t2) x =
   ```

2. Définir la fonction `(codeChar t c)` qui calcule le codage du caractère `c` selon l'arbre de `Huffman t` :

   ```
   > codeChar ht 'e'  ==> [Gauche,Gauche]
   > codeChar ht 'i'  ==> [Droite,Droite,Gauche,Gauche]
   > codeChar ht 'q'  ==> [Droite,Gauche,Gauche,Gauche,Gauche,Droite,Droite]
   ```

   Indications

   ```
   codeChar :: ? -> ? -> Chemin
   codeChar (Tip y) x =
   codeChar (Bin t1 t2) x
           | appartient t1 x =
           |
   ```

3. En déduire la la fonction `(code t xs)` qui calcule le codage de la chaîne `xs` selon l'arbre de `Huffman t` :

   ```
   > code ht "ei" ==> [Gauche,Gauche,Droite,Droite,Gauche,Gauche]
   > code ht "eie" ==> [Gauche,Gauche,Droite,Droite,Gauche,Gauche,Gauche,Gauche]
   ```

4. Vérifier que les caractères fréquents ont un code beaucoup plus court que les caractères peu fréquents.

   ```
   > length (code ht "tete")  ==> 12
   > length (code ht "kwkw")  ==> 48

   > code ht "tete"
   ==> [Droite,Droite,Droite,Droite,Gauche,Gauche,Droite,Droite,Droite,Droite,Gauche,Gauche]

   > code ht "kwkw"
   ==> [Droite,Gauche,Gauche,Gauche,Gauche,Droite,Gauche,Droite,Gauche,Gauche,Droite,Gauche,
   Droite,Gauche,Gauche,Gauche,Gauche,Droite,Gauche,Droite,Gauche,Gauche, Droite,Droite,Droite,
   Gauche,Gauche,Gauche,Gauche,Droite,Gauche,Droite, Gauche,Gauche, Droite,Gauche,Droite,Gauche,
   Gauche,Gauche,Gauche,Droite,Gauche,Droite,Gauche,Gauche,Droite,Droite]
   ```

2 Décodage

Compléter la définition de la fonction `(decode t ps)` qui, à partir d'un arbre de `Huffman t` et d'un chemin `ps`, renvoie la chaîne de caractères dont le code est `ps`.

```
> decode ht [Gauche,Gauche,Droite,Droite,Gauche,Gauche,Gauche,Gauche] ==> "eie"
> decode ht [Gauche,Gauche,Droite,Droite,Gauche,Gauche] ==> "ei"
> decode ht [Gauche,Gauche,Droite,Gauche,Gauche,Gauche,Gauche,Droite,Droite] ==> "eq"

decode :: Tree Char -> Chemin -> String
decode t ps = trace t t ps

trace :: Tree Char -> Tree Char -> Chemin -> String
trace t (Tip x) [] = ?
trace t (Tip x) (p:ps) = x: trace t ? ?
trace t (Bin t1 t2) (Gauche:ps) = trace t ? ?
trace t (Bin t1 t2)      ?      = trace t ? ?
```

Indications : A quoi peut servir le premier paramètre `t` *de la fonction* `trace` *? Pourquoi reste-t-il inchangé d'un appel à l'autre ?*

3 Expérimentations

1. La fonction `(calculFrequence xs)` détermine, pour chaque caractère de `xs`, son nombre d'occurrences dans `xs`. La liste résultat est ordonnée par valeurs croissantes du nombre d'occurrences. Définir les fonctions `ajouteFrequence` et `tri`.

   ```
   > calculFrequence "azeeez"  ==> [('a',1),('z',2),('e',3)]
   > calculFrequence "azertyzertert"  ==> [('y',1),('a',1),('z',2),('t',3),('r',3),('e',3)]

   calculFrequence :: String -> Frequences
   calculFrequence xs = tri (calculFrequence2 xs [])

   calculFrequence2 :: String -> Frequences -> Frequences
   calculFrequence2 "" fs = fs
   calculFrequence2 (c:cs) fs = calculFrequence2 cs (ajouteFrequence c fs)
   ```

2. En déduire la fonction `(codeText xs)` qui calcule le codage de `xs` (en construisant l'arbre de `Huffman` associé à `xs`).
   Les valeurs `Gauche` et `Droite` seront transformées en `'0'` et `'1'`.

   ```
   > codeText "hello world"  ==> "00100010101110110101111101101100"
   > codeText "un corbeau sur un arbre"
      ==> "11110111101000100110110100010001111101001011101110111011110000011010010001"
   ```

3. Définir la fonction `(tauxDeCompression xs)` qui permet de comparer la taille du code `ASCII` de `xs` avec la taille du code `Huffmann` de `xs`.
   Pour convertir un entier en un flottant, on pourra utiliser la fonction `fromIntegral`.

   ```
   > tauxDeCompression "un corbeau sur un arbre"  => 0.5978261
   > tauxDeCompression "aaaaabbbbb"  ==> 0.875
   ```

4 Construction de l'arbre de `Huffman`

On définit les types synonymes `Frequence` et `Frequences` comme suit :

```
data Tree a = Tip a | Bin (Tree a) (Tree a)
    deriving Show

type Frequence = (Char, Integer)

type Frequences = [Frequence]
```

1. Le poids d'un arbre de fréquences est défini comme étant la somme des fréquences de chacune de ses feuilles. Définir la fonction `(poids t)` qui calcule le poids d'un arbre de fréquence `t`.

   ```
   > poids (Bin (Tip ('s',332)) (Tip ('t',360)))  ==> 692
   > poids (Bin (Bin (Tip ('i',277)) (Tip ('n',284))) (Tip ('s',332)))  ==> 893
   ```

2. Définir la fonction `(unlabel tf t)` qui transforme un arbre de fréquences `tf` en un arbre de caractères `t` : *(Indications : diapos #6, 7 et 8 du CM #6)*

   ```
   > unlabel (Tip ('i',277))
         ==> (Tip 'i')
   > unlabel (Bin (Tip ('s',332)) (Tip ('t',360)))
         ==> (Bin (Tip 's') (Tip 't'))
   > unlabel (Bin (Bin (Tip ('i',277)) (Tip ('n',284))) (Tip ('s',332)))
         ==> (Bin (Bin (Tip 'i') (Tip 'n')) (Tip 's'))
   ```

3. Compléter la définition de la fonction `(insert u ts)` qui insère un arbre de fréquences `u` dans une liste d'arbres de fréquence `ts`. Cette liste est ordonnée selon l'ordre croissant de leur poids.

   ```
   insert :: Tree Frequence -> [Tree Frequence] -> [Tree Frequence]
   insert u []                = ?
   insert u (t:ts)
          | poids u <= poids t = ?
          | otherwise          = ?
   ```

4. Définir la fonction `(single ts)` qui détermine si une liste `ts` est réduite à un seul élément.

   ```
   > single [(Bin (Bin (Tip 'i') (Tip 'n')) (Tip 's')), (Tip 'a')]  ==> False
   > single [(Bin (Bin (Tip 'i') (Tip 'n')) (Tip 's'))]  ==> True
   ```

5. La fonction prédéfinie `(until p f e)` applique la fonction `f` à `e` jusqu'à ce que `(p e)` soit vrai. Que fait la fonction `build` ?
   Vérifier que le type de la fonction `until` est : `(a -> Bool) -> (a -> a) -> a -> a`.

   ```
   build :: Frequences ->  Tree Char
   build = unlabel . head . until single combine . (map (\x->Tip x))

   combine :: [Tree Frequence] -> [Tree Frequence]
   combine (t1:t2:ts) = insert (Bin t1 t2) ts
   ```