Programmation IV 4 avril 2007

Second mini-projet : codages de Huffman

Un codage de Huffman est un algorithme de compression des chaînes de caractères. Dans un texte non-compressé, chaque caractère est représenté par le même nombre de bits (souvent huit). Un codage de Huffman assigne à chaque caractère une représentation de longueur différente, suivant s'il est commun ou non. Bien entendu, un codage donné n'est optimal que pour un seul texte.

Un codage de Huffman peut être représenté par un arbre binaire dont les feuilles sont les caractères à encoder. Chaque noeud est un ensemble contenant les caractères présents dans les feuilles en dessous. De plus, chaque caractère dispose d'un poids — sa fréquence dans le texte — et chaque noeud est annoté du poids total des feuilles en dessous, une information nécessaire lors de la construction de l'arbre. La figure 1 est un arbre de codage pour un texte où "A" à la fréquence 8, "B" la fréquence 3 et tous les autres caractères la fréquence 1.

Pour un arbre de codage donné, on obtient l'encodage d'un caractère en traversant l'arbre de la racine à la feuille contenant le caractère. En chemin, chaque fois que l'on l'on prend une branche à gauche, on ajoute 0 à la représentation, chaque fois que l'on prend une branche à droite, on ajoute 1. L'arbre de la figure 1 encode le caractère "D" comme "1011".

Le décodage commence à la racine de l'arbre. On lit successivement les bits de la séquence à décoder ; pour chaque 0 on descend dans l'arbre à gauche, pour chaque 1 à droite. Lorsqu'on atteint une feuille, on a décodé le caractère correspondant et l'on recommence à la racine. La séquence de bits "10001010" se décode avec l'arbre de la figure 1 en "BAC".

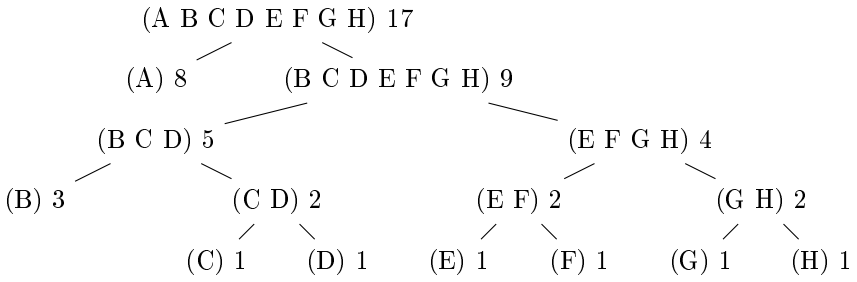

FIG. 1 – Un codage de Huffman encodé comme un arbre

Mise en place

En Scala, un arbre de codage de Huffman est représenté par le type algébrique suivant.

```
abstract class CodeTree
case class Fork (
  left: CodeTree,
```

```
  right: CodeTree,
  characters: List[Char],
  weight: Int
) extends CodeTree
case class Leaf(character: Char, weight: Int) extends CodeTree
```

Vous devez ...

- définir "`weight`" qui retourne la poids total d'un arbre de codage donné.

  ```
  def weight(tree: CodeTree): Int = ...
  ```

- définir "`chars`" qui retourne la liste des caractères définis dans un arbre de codage donné.

  ```
  def chars(tree: CodeTree): List[Char] = ...
  ```

Une fois définies, ces fonctions vous permettent d'utiliser "`makeCodeTree`" qui facilite la création d'un arbre de codage en calculant automatiquement la liste des caractères et le poids lors de la création d'un noeud.

```
def makeCodeTree(left: CodeTree, right: CodeTree) =
  Fork (
    left, right, chars(left) ::: chars(right),
    weight(left) + weight(right)
  )
```

"`makeCodeTree`" s'utilise de la façon suivante.

```
val sampleTree = makeCodeTree (
  makeCodeTree(Leaf('x', 1), Leaf('e', 1)),
  Leaf('t', 2)
)
```

Décodage

Vous devez définir "`decode`", une fonction capable de décoder une liste de bits encodés par un codage de Huffman, étant donné l'arbre de codage correspondant (`type Bit = int`).

```
def decode(tree: CodeTree, bits: List[Bit]): List[Char] = ...
```

Encodage ...

Cette section concerne l'encodage de Huffman d'une séquence de caractères en un séquence de bits. La second partie — par table de codage — est plus difficile que la première : vous aurez peut-être intérêt à l'aborder après avoir résolu la section concernant la création des arbres de codage.

...par arbre de Huffman

Vous devez définir "`encode`", une fonction capable d'encoder une liste de caractères à l'aide d'une encodage de Huffman, étant donné un arbre de codage.

```
def encode(tree: CodeTree)(chars: List[Char]): List[Bit] = ...
```

Votre implantation traversera l'arbre de codage pour chaque caractère, une tâche que vous avez tout intérêt à abstraire dans une fonction annexe.

. . .par table de codage

La fonction précédente est simple, mais peu efficace. Vous devez maintenant définir une nouvelle fonction "`fastEncode`" fonctionnellement équivalente, mais plus efficace.

```
def fastEncode(tree: CodeTree)(chars: List[Char]): List[Bit] = ...
```

Votre implantation construira une seule fois une table des caractères qui, pour chaque caractère possible fait correspondre la liste de bits l'encodant. La façon la plus simple — mais pas la plus efficace — est d'encoder la table de caractères comme une liste de pairs.

```
type CodeTable = List[Pair[Char, List[Bit]]]
```

L'encodage se fait ensuite en accédant à cette table, par exemple à l'aide de la fonction "`lookup`".

```
def lookup(table: CodeTable, character: Char): List[Bit]
```

La création de la table est définie par "`transform`" qui traverse l'arbre de codage en construisant la table des caractères au fur et à mesure.

```
def transform(t: CodeTree): CodeTable = t match {
  case Leaf(c, _) => ...
  case Fork(l, r, _, _) =>
    mergeCodeTables(transform(l), transform(r))
}
def mergeCodeTables(a: CodeTable, b: CodeTable): CodeTable = ...
```

Création d'arbres de codage

Jusqu'à présent, l'arbre de codage était fixe. Utiliser le même arbre pour tous les encodages permet d'éviter de devoir le transmettre avec les données. Cet arbre est construit en fonction de la fréquence des caractères en français et donnera de bons résultats moyens pour les textes dans cette langue.

Toutefois, étant donné un texte, il est possible de calculer un arbre de codage optimal, dans le sens où l'encodage de ce texte sera de longueur minimum — en gardant toute l'information. Vous devez définir "`optimalCodeTree`", une fonction qui calcule un tel arbre optimal à partir d'une liste de caractères.

```
def optimalCodeTree(chars: List[Char]): CodeTree = ...
```

- Pour commencer, calculez la fréquence de chaque caractère dans le texte.

  ```
  def freqs(chars: List[Char]): List[Pair[Char, Int]]
  ```

- Construisez ensuite une liste contenant toutes les feuilles de l'arbre de codage à construire — le cas "`Leaf`" du type algébrique "`CodeTree`" — ordonnées par leur poids, c'est-à-dire par la fréquence du caractère.
- Enlevez les deux arbres de poids le plus faible de la liste construite à l'étape précédente. Fusionnez-les en un seul arbre par un "`Fork`". Ajoutez ce nouvel arbre à la liste (maintenant plus courte d'un élément) en maintenant l'ordre. Continuez de cette façon jusqu'à ce que la liste ne contienne plus qu'un seul arbre. Cet arbre est l'arbre de codage optimal.