Projet : compression LZW

Version du 9 novembre 2016

Vous pouvez (et devez) découvrir les algorithmes de compression et décompression de Lempel-Ziv-Welch sur la page de Wikipedia :

https://fr.wikipedia.org/wiki/Lempel-Ziv-Welch

Ces algorithmes se basent sur l'idée de disposer d'une table de traduction entre des chaînes de caractères et un ensemble de codes (les codes peuvent par exemple être des entiers).

Compression

On démarre avec une table contenant tous les caractères individuellement, avec leurs traductions. On parcourt un flux de caractères, en lisant dans le flux, le plus long préfixe w contenu dans la table. On écrit alors en sortie le code de ce préfixe. Si le flux n'est pas terminé, on itère, mais auparavant on ajoute à la table le mot wc (avec un nouveau code), où c est le prochain caractère du flux.

Décompression

On s'aperçoit qu'on peut décompresser un flux de codes en reconstruisant la table de traduction à fur et mesure. On démarre avec la table contenant tous les caractères (et leur code associé), comme pour la compression.
Quand on lit un code du flux, on cherche son image inverse dans la table, c'est-à-dire la chaîne auquel ce code est attribué, et on produit cette chaîne comme résultat partiel. On itère avec le reste du flux, mais en se rappelant que la construction du flux de codes a étendu la table de traduction en ajoutant la correspondance entre ce mot et le premier caractère du décodage du prochain code à venir.

Il se pose un problème, car en principe nous ne savons pas décoder le prochain code (qui pourrait ne pas être un le code d'un mot dans la table).

Première solution. L'algorithme décrit sur wikipedia propose de résoudre le problème en ajoutant la correspondance entre le mot précèdent et le premier caractère de la traduction du code courant. Nous avons donc besoin d'un « buffer » w où enregistrer le mot précèdent pour l'itération courante. Ainsi, si le flux de code n'est pas terminé, on enregistre un nouveau code, dont la chaîne associée est celle du code précèdent plus la première lettre de la chaîne associée au code courant. Le schéma de décompression, comme décrit, est imparfait car la mise à jour de la table a lieu en retard d'une itération par rapport à la compression ; il se peut donc que l'on trouve pas la traduction (inverse) du code dans la table. Par ailleurs, dans ce cas, la traduction du code courant est nécessairement de la forme wc où w ce le mot dans le buffer et c est aussi la première lettre de w. Donc, au cas que l'on ne trouve pas la traduction du code courant dans la table, on écrit le wc en sortie, on l'ajoute à la table, et on itère avec le reste du flux.

Deuxième solution. (Merci à Guyslain Naves de l'avoir suggérée). Quand on lit un code du flux, on cherche son image inverse dans la table, c'est-à-dire la chaîne auquel ce code est attribué, et on produit cette chaîne comme résultat partiel. On itère avec le reste du flux, mais en se rappelant que la construction du flux de codes a étendu la table de traduction (en ajoutant la correspondance entre ce mot et le premier caractère du décodage du prochain mot à venir).

Ainsi, si le flux de code n'est pas terminé, on enregistre un nouveau code, dont la chaîne associée est celle du code actuel plus la première lettre de la chaîne associée au code suivant. Attention, il se peut que le code suivant soit celui en train d'être enregistré (essayez avec le mot `aaa`). Ensuite on continue l'itération.

Objectif. Écrire deux programmes pour compresser/décompresser des fichiers binaires en utilisant la méthode LZW.

1 Un classe bien choisie : Table

Nous vous proposons de démarrer le projet avec une abstraction de ce qu'est une table de traduction. Supposons que nous disposions d'un type `Code` pour représenter les codes (prenez aux début `type Code = Int`). Une table de traduction est un type, quel qu'il soit, ayant les opérations de traduction directe et/ou inverse nécessaire. Cette abstraction revient, en Haskell, à définir une classe :

```
class Table a where
  empty :: a
  insert :: a -> String -> a
  codeOf :: a -> String -> Maybe Code
  stringOf :: a -> Code -> Maybe String
  isIn :: a -> String -> Bool
  split :: a -> String -> (String,Maybe Code,String)
```

La signification des méthodes de cette classe est évidente :

- `empty` est la table de traduction vide,
- `insert` ajoute un mot à une table,
- `codeOf` et `stringOf` sont les deux opérations élémentaires de traduction. `codeOf` vérifie si une mot est dans la table et si oui retourne le code correspondant (notez ici l'utilisation du type `Maybe a`, pour prévoir la réponse "on n'a rien trouvé"). `stringOf` opère la traduction inverse.
- `isIn` indique si un mot est contenu dans la table de traduction.
- La méthode la moins évidente est `split`. Cette fonction prend une table, une chaîne de caractères, et subdivise cette chaîne entre son plus long préfixe contenu dans la table et le suffixe restant. Entre autres, cette fonction renvoie aussi le code de la chaîne de caractères trouvée.

Question 1. En utilisant les méthodes de la classe `Table`, esquissez la définition des fonctions

```
lzwEncode :: Table a => a -> String -> [Code]
lzwDecode :: Table a => a -> [Code] -> String
```

Pour la deuxième fonction, vous vous servirez d'une fonction supplémentaire,

```
lzw_Decode :: Table a => a -> String -> [Code] -> String
```

dont le deuxième paramètre est la chaîne correspondant au dernier code lu. Cette fonction ajoute le nouveau code dans la table, selon la méthode décrite précédemment, et décode la suite du flux.

Vérifiez que vos définitions ne produisent pas d'erreurs de type avant de continuer.

2 Une première implantation par listes associatives

Question 2. Definissez un *nouveau* type de liste associatives (voir par exemple `http://book.realworldhaskell.org/read/data-structures.html`). et instanciez ce type à la classe `Table`.

Question 3. Testez la correction de votre implantation des algorithmes ce compression et décompression avec au moins 20 exemples (ces exemples pourront être engendrés au hasard, voir la bibliothèque QuickCheck. Évidemment il s'agira de montrer que la décompression d'un texte compressé est égale au texte originel.

3 Une deuxième implantation, par les arbres de préfixes

Question 4. Nous avons vu les arbres de préfixes en TP 4. Adaptez cette structure de données afin qu'elle puisse être instanciée à la classe `Table`. Cette structure de données vous permettra de faire les ajouts, les traductions, et évaluer la fonction `split` de façon efficace.

Question 5. Avec les 20 tests, mesurez les performances de l'algorithme selon la structure de donnée utilisée.

4 Des flux de caractères aux flux de bits

Sauvez votre première version du projet (et ajoutez cette version dans le rendu du projet) ; nous allons recommencer.

Après avoir implanté les algorithmes, j'obtiens ces résultats :

```
*Main> length $ lzwEncode chars "TOBEORNOTTOBEORTOBEORNOT"
16
*Main> length "TOBEORNOTTOBEORTOBEORNOT"
24
*Main> 16*16
512
*Main> 25*8
200
```

C'est-à-dire, dans l'hypothèse qu'un caractère soit codé sur 8 bits, et qu'un entier soit codé sur 16, au lieu de diminuer la taille du flux, nous l'avons accrue ! Pour pouvoir dire que nous avons diminué la taille du flux sur ce petit exemple, il faudrait pouvoir encoder les codes sur au plus 12 bits. C'est ce que nous allons faire.

Question 6. Définissez un type `Byte` et un type `Int10`. Le premier représentera une suite de 8 bits, l'autre représentera un entier codé sur 10 bits.

Modifiez la définition de la classe Table (essayez de rendre votre code le plus polymorphe possible) de votre code afin que l'on puisse compresser/décompresser une liste de `Byte` vers une liste de `Int10`.

5 Finition et tests

Pour terminer, on vous demande de produire deux exécutables (utilisez donc `ghc` et/ou `runhaskell` à la place de `ghci`) nommés `lzwCompress` et `lzwDecompress` pour compresser/décompresser des fichier (unix/linux) binaires.

Tests. Les tests (et la validation du code) sont une partie intégrante de n'importe quel projet. Un fichier `README` contiendra une description précise de toute démarche prise pour valider la qualité de votre code. En pratique : pas de tests, moins de points, plus de tests, plus de points.

Remarques

- Le projet se déroule en binômes. Si vous n'arrivez pas à trouver un binôme, veuillez le signaler, un binôme pourra alors vous être attribué.
- Date limite du rendu du projet : le jeudi 30/11/2016, à 23h59 (heure de Paris). Envoyez votre projet **en format compressé `zip` ou `tgz` à votre encadrant de TP**. Le nom du fichier envoyé **indiquera clairement** les deux noms (pas les prénoms) des membres du binômes.
- La note de projet est individuelle, et peut ne pas être égale pour les deux membres du binôme.
- La soutenance est une partie nécessaire de l'évaluation de votre projet ; la présence de chaque individu à la soutenance est donc obligatoire. L'absence à la soutenance implique une note de 0 à votre projet.
- Vous pouvez trouver sur Internet plusieurs sources concernant le codage LZW ; vous pouvez vous en inspirer. Si vous utilisez, dans votre projet, du code dont vous n'êtes pas complètement les auteurs, veuillez le signaler clairement dans le fichier appelé `README.txt` et inclure les sources qui vous ont inspirés dans le projet.
  Par ailleurs :
  - On vous demande de suivre strictement les étapes décrites dans le projet.
  - On testera, lors de la soutenance et pour chaque étape, la compréhension du code dont vous êtes supposés être les auteurs. Une incompréhension manifeste (ou miminale) du code présenté à la soutenance entraînera la note 0 (ou minimale) à votre projet.
- Dates des soutenances :
  - à Luminy : le mardi 6/12/2016 (de 8h00 à 13h00) ;
  - à St-Jérôme : le mercredi 7/12/2016 (de 8h00 à 16h00).