Cycle ingénieur - 2ème année

Programmation fonctionnelle en Scala

Projet *Méthodes de compression sans perte*

2021-2022

Consignes

Consignes générales

Objectif du projet

- Implémenter certaines méthodes de compression sans perte

Consignes générales

- Projet effectué par groupes de **4 à 5 étudiants**
- Date limite de rendu (par mail) : **5 décembre 2021, 23h59**
- Soutenance : semaine du **6 au 10 décembre 2021**

Nature du rendu

- *Fourni* : Projet `sbt` à compléter
- **À faire** : implémenter les parties `???`
- **Rendu** :
  - archive du répertoire `src/main/scala` uniquement
  - rapport court

Consignes spécifiques

Modification du code existant

- Modification **INTERDITE** :
  - des classes déjà implémentées
  - des valeurs déjà implémentées
  - des méthodes déjà implémentées
  - de la signature des valeurs et méthodes à implémenter
- Ajout *autorisé* de nouvelles classes, valeurs et/ou méthodes intermédiaires pour implémenter les éléments demandés

Consignes spécifiques

Tests des valeurs et méthodes à implémenter

- Des tests seront effectués sur votre code.
  Ces tests participent à la note finale du projet.
- Pour fonctionner, **ces tests supposent que les consignes précédentes ont été respectées**.
  Dans le cas contraire, les tests ne compileront pas.

Tests fournis

- Des tests basiques sont fournis (commande `test` dans `sbt`).
- Ces tests permettent de vérifier si les vrais tests compileront.
- *Réussir les tests fournis n'implique pas que les vrais tests seront réussis.*

Consignes spécifiques

Respect des règles de programmation fonctionnelle

- Utilisation des variables (`var`) **interdite**
- Utilisation des boucles (conditionnelles ou non) **interdite**
- ***Favoriser la récursivité terminale***

Quelques conseils

- *for-comprehensions* (`for ... yield`) *autorisées*.
- Utiliser le plus possible la bibliothèque des collections Scala

Rapport

Rapport de développement

- Description succincte de la conception mise en place
- Description succincte de l'implémentation mise en place

Étude des méthodes de compression

- Méthode RLE
  - quelles configurations donnent les meilleures performances ?
- Méthodes statistiques :
  - comparaison Huffman / Shannon-Fano
  - quelles configurations donnent les meilleures performances ?
- Méthodes à dictionnaire :
  - comparaison LZ78 / LZW
  - quelles configurations donnent les meilleures performances ?

Analyse de performance de la parallélisation (`.par`) *[Facultatif]*

Soutenance (semaine du 6/12 au 10/12)

Organisation

- Durée totale : *20 minutes*
  - Temps de présentation : **entre 10 et 15 min**

    Contrainte de temps à respecter absolument
  - Reste du temps consacré à nos questions
- **Tous les membres du groupe doivent présenter.**

Attendu

- Présenter la solution conceptuelle
- Présenter ***brièvement*** l'implémentation
- **Présenter un jeu de test pour valider l'implémentation**

Contenu du répertoire `src/main/scala` fourni

Aucune modification n'est requise dans les fichiers en *italique*.

- src/main/scala
  - *Compressor.scala*
  - RLE.scala
  - statistic
    - *Bit.scala*
    - EncodingTree.scala
    - StatisticCompressor.scala
    - Huffman.scala
    - ShannonFano.scala
  - lz
    - *Dictionaries.scala*
    - LZ78.scala
    - LZW.scala

Présentation du projet

Présentation du projet

Présentation générale

Présentation générale

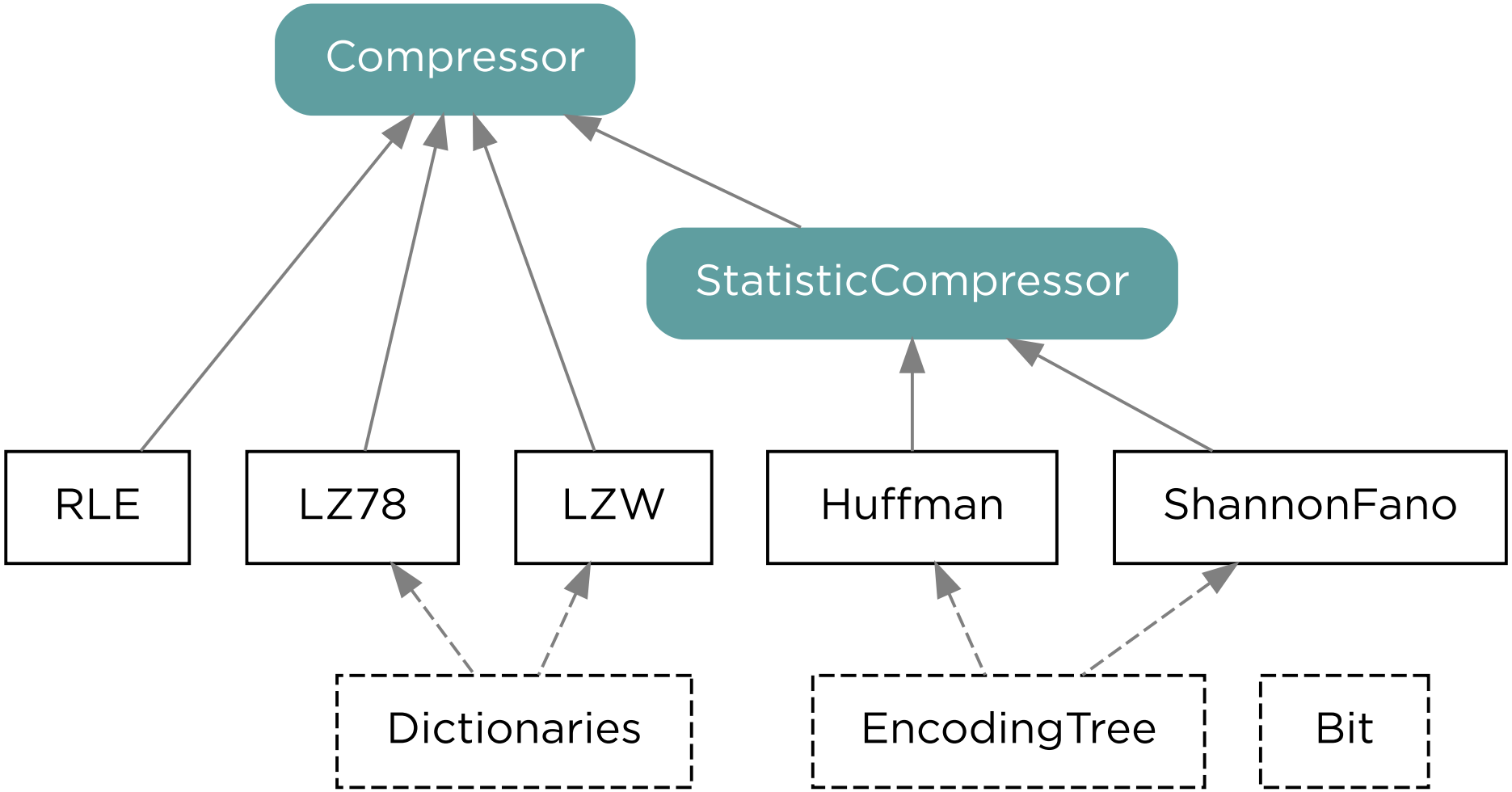
Motivation de la compression sans perte

- Réduire le temps de transfert en réduisant la taille
  ⇒ algorithme de compression
- Résultat de l'algorithme souvent peu lisible
  ⇒ algorithme de décompression
- Compression **sans perte** : la décompression doit reconstruire les données originelles ***à l'identique***

Méthodes étudiées dans ce projet

- RLE *(Run-Length Encoding)*
- Méthodes statistiques : Huffman, Shannon-Fano
- Méthodes à dictionnaire : LZ78, LZW

Hiérarchie des types du code fourni



`Compressor[S, C]`

Définit le cadre général d'une méthode de compression

Données manipulées

- Séquence de symboles
- `S` : type d'un symbole
- *Cas usuel : chaînes de caractères*

  `String` ≡ `Seq[Char]` ⇒ `S` = `Char`

  (une conversion explicite sera peut-être nécessaire)

Résultat de la compression

- `C` : type du résultat
- *dépend fortement de la méthode utilisée*

`Compressor[S, C]`

Méthodes du trait `Compressor`

Une méthode de compression définit donc deux opérations :

- `def compress(msg : Seq[S]) : C`
  la *compression* transforme la séquence de symboles en un objet compressé
- `def uncompress(res : C) : Option[Seq[S]]`
  la *décompression* effectue (si possible) l'opération inverse pour retrouver la séquence originale **à l'identique**

Autrement dit, si `A` hérite de `Compressor`, alors pour toute instance a de `A` :

```
a.uncompress(a.compress(msg)) == Some(msg)
```

Présentation du projet

RLE

RLE *(Run-Length Encoding)*

La méthode consiste à « compresser » les symboles identiques qui se suivent.

Exemples

- `""` → `Seq()` (séquence vide)
- `"a"` → `Seq((a, 1))`
- `"aaaabbcbbb"` → `Seq((a, 4), (b, 2), (c, 1), (b, 3))`

À implémenter dans `src/main/scala/RLE.scala`

- `compress` : méthode de compression RLE
- `uncompress` : méthode de décompression RLE

Présentation du projet

Méthodes statistiques

Méthodes statistiques

Ces méthodes s'appuyant sur la connaissance de la distribution des symboles dans la séquence d'entrée (appelée *source*).

On commence donc par calculer cette distribution et quelques indicateurs.

À implémenter dans `src/main/scala/statistic/StatisticCompressor.scala`

- `occurrences` : tableau associatif donnant, pour chaque symbole présent, le nombre de fois qu'il apparaît dans la source
- `entropy` : l'entropie de la source, i.e. $\texttt{entropy} = -\sum_s \frac{n(s)}{N} \cdot \log_2\left(\frac{n(s)}{N}\right)$
  - N : longueur de la source
  - $n(s)$: nombre d'apparitions du symbole s dans la source
- `orderedCounts` : séquence des couples (symbole, effectif) triée par ordre croissant du nombre d'apparitions dans la source

Méthodes statistiques : Huffman / Shannon-Fano

Résultat de la compression

- Chaque symbole est représenté par une suite de bits

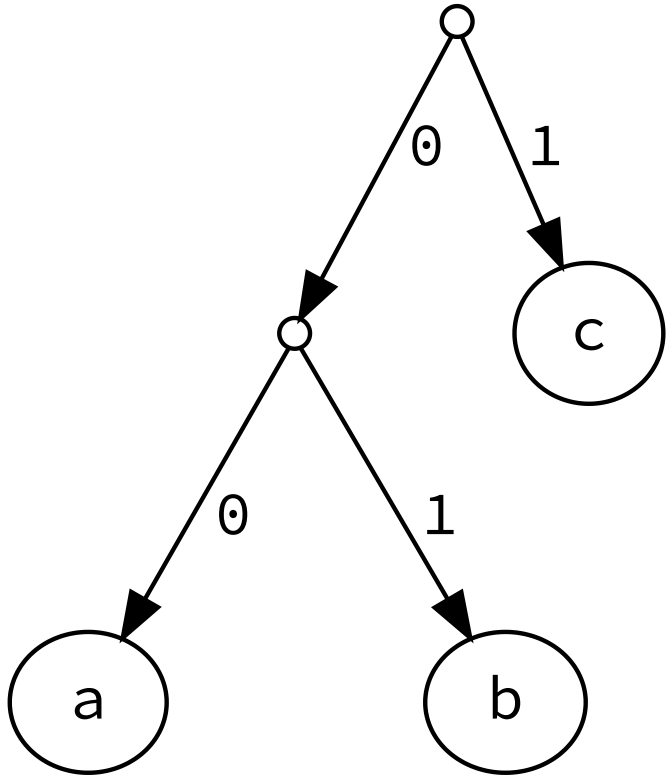
  ⇒ résultat : séquence de bits

  ⇒ `StatisticCompressor[S]` hérite de `Compressor[S, Seq[Bit]]`
- Type `Bit` fourni avec :
  - la constante `Zero` (pour `0`)
  - la constante `One` (pour `1`)

Représentation du codage binaire

- Chaque méthode définit donc un codage binaire de chaque symbole.
- Un tel codage est représenté sous la forme d'un arbre binaire dont les feuilles sont les symboles apparaissant dans la source.

Arbre binaire de codage



Symbole	'a'	'b'	'c'
Codage binaire	00	01	1

- "a" → 00
- "ab" → 0001
- "abbca" → 000101100

La convention utilisée pour identifier les branches est totalement arbitraire et ne nuit à la performance.

Type `EncodingTree[S]`

Description

Arbre binaire *étiqueté* par des entiers :

- Feuille (`EncodingLeaf[S]`) :
  - `value : S` → valeur de la feuille (symbole)
  - `label : Int` → étiquette ;
- Nœud (`EncodingNode[S]`) :
  - `label : Int` → étiquette ;
  - `left, right : EncodingTree[S]` → sous-arbres gauche et droit

Quelle valeur pour les étiquettes ?

- Feuille : nombre d'apparition du symbole dans la source
- Nœud : somme des étiquettes des sous-arbres

Type `EncodingTree[S]`

Exemple avec la source "abbca" (entropie ≃ 1,52)

Symbole	`'a'`	`'b'`	`'c'`
Codage binaire	`00`	`01`	`1`

Longueur moyenne du code :

$$\frac{2}{5} \cdot 2 + \frac{2}{5} \cdot 2 + \frac{1}{5} \cdot 1 = \frac{9}{5} = \mathbf{1{,}8}$$

Remarques

- D'autres arbres existent (avec des longueurs moyennes différentes).
- **Plus la longueur moyenne est courte, plus le codage est efficace.**
- Limite inférieure théorique *sans perte* : **entropie de la source**

Type EncodingTree[S]

À implémenter dans `src/main/scala/statistic/EncodingTree.scala`

- has : teste si un symbole se trouve dans une feuille de l'arbre
- meanLength : calcule la longueur moyenne du code associée à l'arbre
- encode : transforme un symbole en son code binaire
- decodeOnce : transforme (si possible) les premiers bits en un symbole
- decode : transforme (si possible) la séquence de bits en une séquence de symboles
  ATTENTION : le décodage doit tomber juste avec le dernier bit

Type `StatisticCompressor[S]`

L'arbre (attribut `tree`) sera fourni par les sous-types.

À implémenter dans `src/main/scala/statistic/StatisticCompressor.scala`

- `compress` : encode la séquence de symboles en une séquence de bits avec l'arbre `tree`
- `uncompress` : décode (si possible) la séquence de bits pour retrouver la séquence de symboles

Il reste donc à définir pour chaque méthode la manière de construire l'arbre de codage.

Type `Huffman[S]`

Principe de construction de l'arbre de codage

1. Trier la distribution des symboles par ordre décroissant du nombre d'apparitions
2. Fusionner les deux dernières colonnes et les réinsérer en conservant l'ordre décroissant
3. Recommencer l'étape 2 jusqu'à obtenir une seule colonne
4. Construire l'arbre en inversant les fusions successives

À implémenter dans `src/scala/main/statistic/Huffman.scala`

- `tree` : calcule l'arbre d'encodage selon cette méthode

Type Huffman[S]

Exemple avec la source "abbca" (entropie ≃ 1,52)

1.

Symbole	a	b	c
Effectif	2	2	1

2.

Symbole	bc	a
Effectif	3	2

3.

Symbole	abc
Effectif	5

a	b	c
0	10	11

Longueur moyenne :

$$\frac{2}{5} \cdot 1 + \frac{2}{5} \cdot 2 + \frac{1}{5} \cdot 2 = \frac{8}{5} = \mathbf{1,6}$$

Type ShannonFano[S]

Principe de construction de l'arbre de codage

1. Trier la distribution des symboles rangée par ordre décroissant du nombre d'apparitions
2. Couper en deux sous-distributions *les plus équilibrées possibles*
3. Recommencer l'étape 2 sur chaque sous-distribution jusqu'à obtenir des distributions à un symbole
4. Construire l'arbre à partir des coupes

Les performances ne sont pas meilleures que celles de Huffman.

À implémenter dans `src/scala/main/statistic/ShannonFano.scala`

- `tree` : calcule l'arbre d'encodage selon cette méthode

Type ShannonFano[S]

Exemple avec la source "abbca" (entropie ≃ 1,52)

1.

a	b	c
2	2	1

2.

a	b	c
2	2	1

3.

a	b	c
2	2	1

a	b	c
0	10	11

Longueur moyenne :

$$\frac{2}{5} \cdot 1 + \frac{2}{5} \cdot 2 + \frac{1}{5} \cdot 2 = \frac{8}{5} = \mathbf{1,6}$$

Présentation du projet

Méthodes à dictionnaire

Méthodes à dictionnaire

Principes généraux

- Encodage des séquences de caractères par des références à des emplacements dans un dictionnaire
- *Dictionnaire construit à partir du texte lui-même* contenant toutes les séquences déjà rencontrées
- Très utilisé dans les logiciels de compression/décompression

Avantages lors de la décompression

- Reconstruction du dictionnaire au fur à mesure ⇒ inutile de le fournir
- Code statistique : nécessite l'arbre d'encodage ⇒ qualité réelle moindre

Méthodes à dictionnaire

Restriction du type de données

- Données manipulées : chaînes de caractères ⇒ `S = Char`
- Résultats de la compression :
  - `LZ78` : `Seq[(Int, Char)]`
  - `LZW` : `Seq[Int]`

Objet `Dictionaries` fourni

- Type `Dictionary` : alias de `IndexedSeq[String]`
- Dictionnaires prédéfinis :
  - dictionnaire vide (pour `LZ78`)
  - table ASCII (pour `LZW`)

Type LZ78

Contenu du dictionnaire

1. Initialement : chaîne vide en position 0
2. Recherche de la chaîne maximale existante
3. Ajout au dictionnaire du couple (entier, caractère) avec
   - numéro de la chaîne maximale trouvée
   - dernier caractère suivant cette chaîne

À implémenter dans `src/main/scala/LZ78.scala`

- `compress` : méthode de compression LZ78
- `uncompress` : méthode de décompression LZ78

Type LZ78

Compression de "belle echelle !"

1. "b" en position 1 du dictionnaire, ajout de (0,'b')
2. "e" en position 2 du dictionnaire, ajout de (0,'e')
3. "l" en position 3 du dictionnaire, ajout de (0,'l')
4. "le" en position 4 du dictionnaire, ajout de (3,'e')
5. "␣" en position 5 du dictionnaire, ajout de (0,'␣')
6. "ec" en position 6 du dictionnaire, ajout de (2,'c')
7. "h" en position 7 du dictionnaire, ajout de (0,'h')
8. "el" en position 8 du dictionnaire, ajout de (2,'l')
9. "le␣" en position 9 du dictionnaire, ajout de (4,'␣')
10. "!" en position 10 du dictionnaire, ajout de (0,'!')

→ Seq((0,'b'), (0,'e'), (0,'l'), (3,'e'), (0,'␣'), (2,'c'), (0,'h'), (2,'l'), (4,'␣'), (0,'!'))

Type LZ78

Décompression

1. `(0,'b')` = `"b"` ajouté en position 1 dans le dictionnaire
2. `(0,'e')` = `"e"` ajouté en position 2 dans le dictionnaire
3. `(0,'l')` = `"l"` ajouté en position 3 dans le dictionnaire
4. `(3,'e')` = `"le"` ajouté en position 4 dans le dictionnaire
5. `(0,'␣')` = `"␣"` ajouté en position 5 dans le dictionnaire
6. `(2,'c')` = `"ec"` ajouté en position 6 dans le dictionnaire
7. `(0,'h')` = `"h"` ajouté en position 7 dans le dictionnaire
8. `(2,'l')` = `"el"` ajouté en position 8 dans le dictionnaire
9. `(4,'␣')` = `"le␣"` ajouté en position 9 dans le dictionnaire
10. `(0,'!')` = `"!"` ajouté en position 10 dans le dictionnaire

→ `"belle echelle !"` ***(même dictionnaire que pendant la compression)***

Type LZW

Variante de la méthode LZ78

- Dictionnaire initial non vide
  (Par défaut, table ASCII de 0 à 255)
- À l'apparition d'une nouvelle séquence :
  - on ajoute la chaîne maximale au résultat ;
  - on reprend la lecture **depuis le nouveau caractère**.

À implémenter dans `src/main/scala/LZW.scala`

- `compress` : méthode de compression LZW
- `uncompress` : méthode de décompression LZW

Type LZW

Compression de "belle echelle" (table ASCII de 0 à 255)

1. "be" en position 256 du dictionnaire, ajout de 98 ('b')
2. "el" en position 257 du dictionnaire, ajout de 101 ('e')
3. "ll" en position 258 du dictionnaire, ajout de 108 ('l')
4. "le" en position 259 du dictionnaire, ajout de 108 ('l')
5. "e␣" en position 260 du dictionnaire, ajout de 101 ('e')
6. "␣e" en position 261 du dictionnaire, ajout de 32 ('␣')
7. "ec" en position 262 du dictionnaire, ajout de 101 ('e')
8. "ch" en position 263 du dictionnaire, ajout de 99 ('c')
9. "he" en position 264 du dictionnaire, ajout de 104 ('h')
10. "ell" en position 265 du dictionnaire, ajout de 257
11. "le", ajout de 259

→ `Seq(98, 101, 108, 108, 101, 32, 101, 99, 104, 257, 259)`

Type LZW

Décompression

1. `98 ('b')` : ajout de `"b"`
2. `101 ('e')` : ajout de `"e"`, `"be"` en position 256 du dictionnaire
3. `108 ('l')` : ajout de `"l"`, `"el"` en position 257 du dictionnaire
4. `108 ('l')` : ajout de `"l"`, `"ll"` en position 258 du dictionnaire
5. `101 ('e')` : ajout de `"e"`, `"le"` en position 259 du dictionnaire
6. `32 ('␣')` : ajout de `"␣"`, `"e␣"` en position 260 du dictionnaire
7. `101 ('e')` : ajout de `"e"`, `"␣e"` en position 261 du dictonnaire
8. `99 ('c')` : ajout de `"c"`, `"ec"` en position 262 du dictionnaire
9. `104 ('h')` : ajout de `"h"`, `"ch"` en position 263 du dictionnaire
10. `257` : ajout de `"el"`, `"he"` en position 264 du dictionnaire
11. `259` : ajout de `"le"`

→ `"belle echelle"` ***(même dictionnaire que pendant la compression)***