Chiffrement

3h : avec documents

Année 2020-2021

Préambule

— Les contrats et tests unitaires de la plupart les fonctions vous sont donnés. Vous pouvez compléter les tests unitaires si vous en ressentez le besoin. Les contrats et tests unitaires des fonctions auxiliaires que vous auriez besoin d'ajouter doivent être fournis.
— Pensez à dé-commenter les tests unitaires des fonctions que vous écrivez au fur et à mesure pour tester vos fonctions.
— Les noms et types de modules et fonctions doivent être respectés.
— Pour tester dans `utop`, vous devez ouvrir les modules Chiffrement et Dr (**open** Be.Chiffrement ;; et **open** Be.Dr;;) pour avoir accès aux modules et interfaces que vous définirez.
— Le code rendu doit impérativement compiler (lorsque vous avez une partie de code qui ne compile pas, mettez là en commentaire).

1 Sujet d'étude

1.1 Cadre général

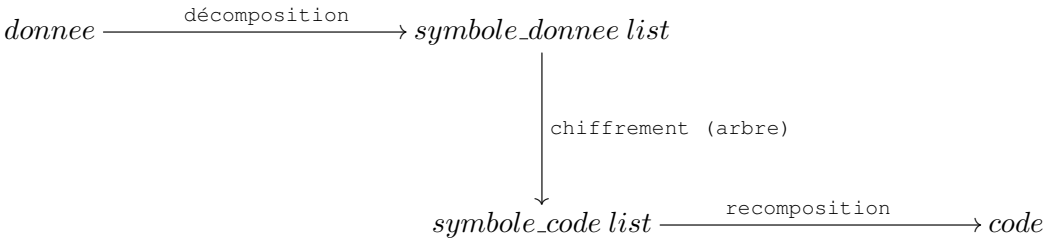
Le but du sujet est d'utiliser les arbres pour chiffrer des données. Pour cela nous manipulerons quatre types de données :

— $donnee$: le type des données à chiffrer (par exemple string)
— $symbole\_donnee$: le type des symboles de l'alphabet de $donnee$ (par exemple char)
— $code$: le type des codes (par exemple int)
— $symbole\_code$: le type des symboles de l'alphabet de $code$ (par exemple chiffre)

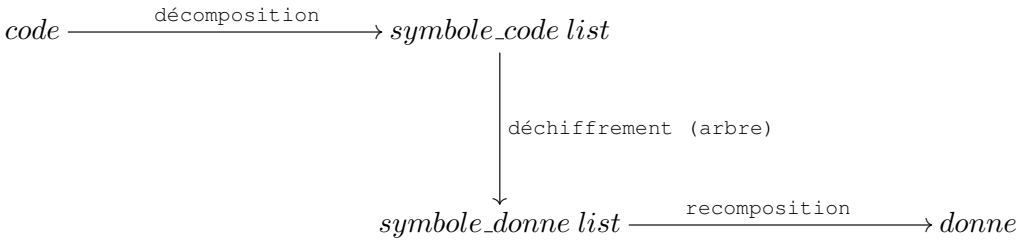
Pour chiffrer une donnée, un arbre de chiffrement contenant des feuilles de type $symbole\_donnee$ et des données de type $symbole\_code$ dans les branches est utilisé. Le chemin pour atteindre un symbole de type $symbole\_donnee$ depuis la racine donne son code ($symbole\_code\ list$). Le processus de chiffrement utilisé est le suivant :

1. la donnée à chiffrer ($donnee$) est décomposée en une liste de symboles de son alphabet ($symbole\_donnee\ list$) ;
2. les codes de chacun des symboles sont concaténés ($symbole\_code\ list$) ;
3. la liste obtenue est recomposée pour obtenir le code ($code$).

chiffrement d'une donnée



Déchiffrement d'une donnée



Lien avec le travail demandé. Entre les types $donnee$ et $symbole\_donnee$, tout comme entre les types $symbole\_code$ et $code$, il est nécessaire d'avoir des fonctions de décomposition et recomposition. C'est le sujet de l'exercice 1.
Entre les types $symbole\_donnee$ et $symbole\_code$, il est nécessaire d'avoir des fonctions de chiffrement et déchiffrement. C'est le sujet de l'exercice 2 et 5 (bonus).
Les exercices 3 et 4 permettent de faire la "glu" entre les exercices 1 et 2.

1.2 Exemples

1.2.1 Un texte chiffré par un entier

Si nous souhaitons chiffrer un texte par un entier, nous pourrons choisir :

- $donnee$ = string (on veut chiffrer un texte)
- $symbole\_donnee$ = char (le texte sera découpé en une suite de caractères)
- $symbole\_code$ = int (chaque caractère sera chiffré par un entier)
- $code$ = int (la donnée sera chiffrée à l'aide d'un entier)

En utilisant l'arbre de chiffrement suivant :

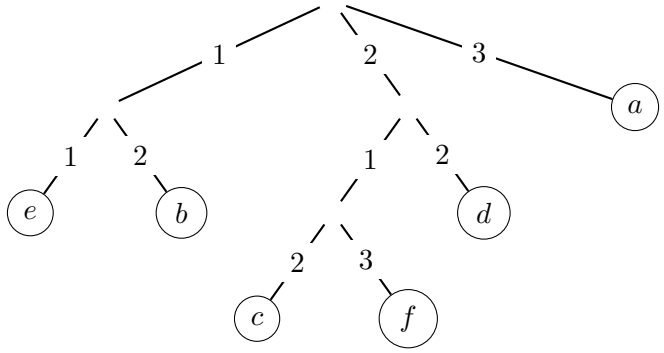


FIGURE 1 – arbre1

'a' se chiffre 3, 'b' se chiffre 12, 'c' se chiffre 212, . . . Le mot "bac" se chiffre 123212.

Mais en utilisant

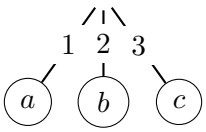


FIGURE 2 – arbre2

le mot "bac" se chiffre 213.

1.2.2 Un texte chiffré par un autre texte

Si nous souhaitons chiffrer un texte par un autre texte, nous pourrons choisir :

- $donnee$ = string
- $symbole\_donnee$ = char
- $symbole\_code$ = char
- $code$ = string

En utilisant l'arbre de chiffrement suivant :

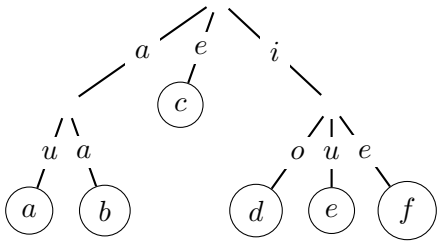


FIGURE 3 – arbre3

'a' se chiffre "au", 'b' se chiffre "aa", 'c' se chiffre "e", ...Le mot "bac" se chiffre "aaaue".

Mais en utilisant (décalage des lettres)

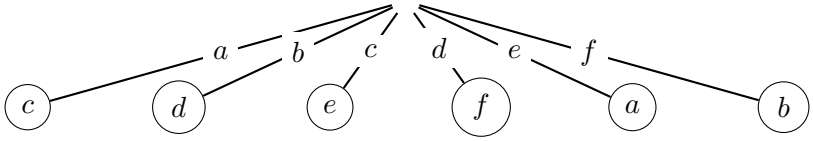


FIGURE 4 – arbre4

le mot "bac" se chiffre "fea".

2 Travail à réaliser

▷ **Exercice 1** *A réaliser dans le fichier* `dr.ml`.
L'interface DecomposeRecompose *possédant les fonctions de décomposition et recomposition vous est fournie.*

1. *Écrire un module* DRString, *conforme à* DecomposeRecompose, *permettant de passer d'une chaîne de caractères à une liste de caractères et inversement. Les types* mot *et* symbole *devront être visibles à l'extérieur du module. Le corps des fonctions* decompose *et* recompose *ainsi que leur contrat et tests unitaires vous ont été fournis dans le TP4, fichier* `chaines.ml` *qui vous est redonné dans les sources.*
2. *Écrire un module* DRNat, *conforme à* DecomposeRecompose, *permettant de passer d'un entier naturel à une liste d'entiers compris entre 0 et 9 et inversement. Par exemple* decompose 248 = [2;4;8] *et* decompose 0 = [0]. *Pour toute liste* l *et entier* e *:* decompose e = l $\Leftrightarrow$ recompose l = e. *Les types* mot *et* symbole *devront être visibles à l'extérieur du module. Vous donnerez les contrats et tests unitaires des fonctions* decompose *et* recompose.

▷ **Exercice 2** *A réaliser dans le fichier* `chiffrement.ml`.
A l'image de ce qui a été fait dans le TP sur les arbres lexicographique, il y a deux modules dans le fichier `chiffrement.ml` *:*

— *le module* Arbre *qui ne travaille que sur l'arbre de chiffrement ;*
— *le module* Chiffrement *qui travaille sur la chaîne complète, donc en intégrant les fonctions de décomposition et recomposition.*

Dans cette exercice vous travaillez sur le module Arbre.

1. *Définissez le type* arbre_chiffrement *, sachant que le code OCaml des 4 arbres du sujet vous sont donnés (*arbre1*,* arbre2*,* arbre3 *et* arbre4*). Pensez à dé-commenter* arbre1*,* arbre2*,* arbre3 *et* arbre4 *une fois votre type* arbre_chiffrement *défini.*
2. *Écrire la fonction* get_branche *qui cherche dans un arbre de chiffrement le sous-arbre associé à un symbole de l'alphabet des codes. Vous pourrez utiliser la fonction* List . assoc_opt *donc le contrat vous est rappelé en annexe.*
3. *Écrire la fonction* dechiffrer *qui prend un code (sous forme de liste de symboles de l'alphabet des codes) et un arbre de chiffrement et renvoie la donnée associée (sous forme de liste de symboles de l'alphabet des données).*
4. *Écrire une fonction* arbre_to_liste *qui renvoie, pour un arbre de chiffrement donné, la liste qui associe à un symbole de l'alphabet des données son code (liste de symboles de l'alphabet des codes).*
5. *Écrire une fonction* chiffrer *qui prend une donnée (sous forme de liste de symboles de l'alphabet des données) et un arbre de chiffrement et renvoie le code associé (sous forme de liste de symboles de l'alphabet des codes).*

▷ **Exercice 3** *A réaliser dans le fichier* `chiffrement.ml`*, module* Chiffrement.
Le squelette du foncteur Chiffrement *vous est donnée. Il est paramétré par deux modules de type* DecomposeRecompose *: l'un pour les données à chiffrer, l'autre pour les codes.*

1. *Compléter le module en spécifiant les valeurs des types* donnee*,* symbole_donnee*,* symbole_code *et* code*.*
2. *Écrire la fonction* dechiffrer *qui prend un code et un arbre de chiffrement et renvoie la donnée associée.*
3. *Écrire une fonction* chiffrer *qui prend une donnée et un arbre de chiffrement et renvoie le code associé.*

▷ **Exercice 4** *A réaliser dans le fichier* `test.ml`*. Pensez à tout dé-commenter.*

1. *Écrire les modules* TexteCodeInt *et* TexteCodeTexte *correspondant respectivement aux exemples de la section 1.2.1 et 1.2.2.*

▷ **Exercice 5** ***BONUS***

— *Définir un itérateur* fold *sur la structure précédente. Donner son type. Les tests unitaires ne sont pas demandés (sera validé avec les fonctions suivantes).*
Aide : Un fold*, en plus de la structure sur laquelle il itère, prend en paramètre une fonction par constructeur du type (fonction de même arité que le constructeur).*
— *Écrire une fonction* nb_symboles *permettant de calculer le nombre de symboles de l'alphabet des données dans l'arbre de chiffrement, à l'aide de l'itérateur précédent.*
— *Écrire une fonction* symboles *permettant de construire la liste des symboles de l'alphabet des données présents dans l'arbre de chiffrement, à l'aide de l'itérateur précédent.*
— *Écrire une fonction* arbre_to_liste_2 *, de même spécification que* arbre_to_liste *, à l'aide de l'itérateur précédent.*

Annexe

List.assoc_opt

```
(* assoc_opt a l returns the value associated with key a in the list of pairs l.
That is, assoc_opt a [ ...; (a,b); ...] = b if (a,b) is the leftmost binding of a in list l.
Returns None if there is no value associated with a in the list l.*)

val assoc_opt : 'a -> ('a * 'b) list -> 'b option
```