Arbres lexicographiques

Objectifs

— Modélisation des données
— Manipulation d'arbre n-aire

Réalisations attendues

— Indispensable : Exercices 1 à 4
— Bonus : Exercices 5 et 6

Rappel : tester à l'aide de Utop

— `dune utop` – lance utop
— `open Tp.Trie ;;` – charge le module `Tp.Trie` i.e les fonctions contenues dans le fichier `trie.ml`
— `nouveau (fun x->x) (fun x->x) ;;` – par exemple pour tester la fonction `nouveau`
— `exit 0;;` – pour quitter

1 Objectifs

On souhaite écrire une application qui va permettre de manipuler un dictionnaire implémenté sous forme de "trie", structure de donnée déjà étudiée en TD. L'application finale s'exécutera et proposera un menu pour choisir les opérations à réaliser sur le dictionnaire.

Pour la réalisation des tests unitaires en cours de développement (comportement "habituel" par défaut), on peut toujours lancer `dune utop` et `dune runtest` comme dans les TP précédents (ne pas hésiter à démarrer la session Utop par un `open Tp.Trie;;`).

Pour compiler les fichiers OCaml et générer un exécutable :
— `dune build appli_chaines.exe` permet de compiler les fichiers et de générer l'exécutable ;
— `_build/default/appli_chaines.exe` permet de lancer l'exécutable.

Les opérations sélectionnées pour les "tries" sont dans le fichier interface `trie.mli`. Le fichier `trie.ml` est un des fichiers à modifier / compléter pour réaliser l'application. Le squelette fourni l'est uniquement pour que l'application puisse compiler.

Comme nous l'avons vu en TD, les opérations sur les "tries" peuvent être réalisée à trois niveaux :
— sur le "trie" lui-même (arbre n-aire + fonctions de décomposition et recomposition) → dans les fichiers `trie.mli` et `trie.ml` ;
— sur l'arbre n-aire → dans le fichier `arbre.ml` déjà complété avec les fonctions vues en TD ;
— sur une liste de branches vue comme une structure associative → dans le fichier `assoc.ml` déjà complété avec les fonctions vues en TD.

Nous vous fournissons un module qui permettra d'avoir un menu de commandes pour manipuler un dictionnaire. Nous vous donnons l'interface `menu_dico.mli` ainsi que l'implantation `menu_dico.ml` (comportant de nombreux effets de bord dont des affichages et des saisies clavier - et donc pas en programmation fonctionnelle pure).

Nous vous fournissons également un fichier source OCaml `chaines.ml` qui contient les fonctions de décomposition et recomposition des chaînes de caractères, afin de faciliter l'écriture de l'application principale.

Enfin, nous vous fournissons un programme d'application (`appli_chaines.ml`) ; son but est essentiellement de définir le dictionnaire vide et d'appeler le menu avec les bons paramètres.

Le but du TP est donc d'implanter le module `Trie`.

2 Préliminaire

Un arbre lexicographique (ou "trie") est une structure utilisée pour représenter des ensembles dont les éléments admettent une décomposition séquentielle en "caractères" possédant de plus un ordre (par exemple, les entiers admettent une décomposition en suite de chiffres, les chaînes de caractères en suite de caractères, etc).

Un arbre lexicographique contient :

- une fonction de décomposition permettant de transformer un élément en liste de "caractères" ;
- une fonction de recomposition, inverse de la précédente ;
- un arbre n-aire dont les branches représentent les suites constituant les éléments stockés dans l'ensemble, les "caractères" étant rangés dans les branches.

De plus, les branches sont triées de gauche à droite. Pour chaque nœud, il doit être indiqué si la suite lue depuis la racine constitue la décomposition d'un élément ou seulement un préfixe strict d'une telle décomposition.

Pour simplifier l'écriture des algorithmes, il vous est demandé de ranger **les "caractères" dans les branches**. Un exemple d'arbre lexicographique stockant le long des branches les mots du mini-dictionnaire suivant : *bas, bât, de, la, lai, laid, lait, lard, le, les, long* est présenté Fig. 1.

Les fonctions de décomposition et recomposition pour la manipulation de mots et de caractères vous sont fournies.

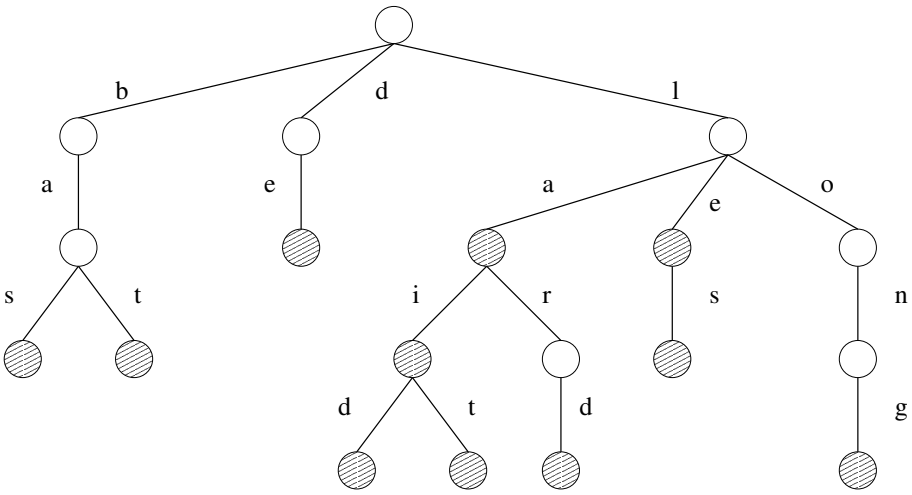

Figure 1 – Un mini-dictionnaire.

3 Manipulation d'arbres

▷ **Exercice 1 (Test d'appartenance)** *Écrire, dans le module* Trie*, la fonction* appartient *qui teste si un élément appartient bien à un ensemble représenté par un arbre lexicographique.*

▷ **Exercice 2 (Retrait)**

- *Écrire, dans le module* Arbre*, la fonction* retrait_arbre *qui enlève un élément d'un arbre n-aire.*
- *Écrire, dans le module* Trie*, la fonction* retrait *qui enlève un élément d'un ensemble.*

Nous souhaitons maintenant parcourir dans l'ordre lexicographique les éléments d'un arbre, parcours idéal pour un dictionnaire. Ceci correspond à un parcours en profondeur des branches de l'arbre.

▷ **Exercice 3 (Parcours)** *Écrire, dans le module* Arbre*, une fonction qui renvoie la liste des éléments dans l'ordre lexicographique d'un arbre passé en paramètre.*

▷ **Exercice 4 (Affichage)** *Écrire, dans le module* Trie*, la fonction* affiche *qui affiche le contenu du dictionnaire.*

4 Normalisation

Le problème de l'opération de retrait est qu'elle peut créer dans l'arbre des branches inutiles, i.e. dont les nœuds ne correspondent jamais à un élément. Ce peut être le cas si on retire les mots *lai*, *laid* et *lait* du dictionnaire de la figure 1.

▷ **Exercice 5 (Normalisation)** *Écrire, dans le module* Arbre*, une fonction qui élimine les branches inutiles d'un arbre n-aire.*

▷ **Exercice 6 (Retrait normalisé)** *Modifier dans le module* Trie*, la fonction de retrait en intégrant le fonctionnement de la fonction de normalisation précédente (ne laisse jamais de branches inutiles dans un arbre lexicographique).*