Programmation Fonctionnelle Avancée

Cours 9

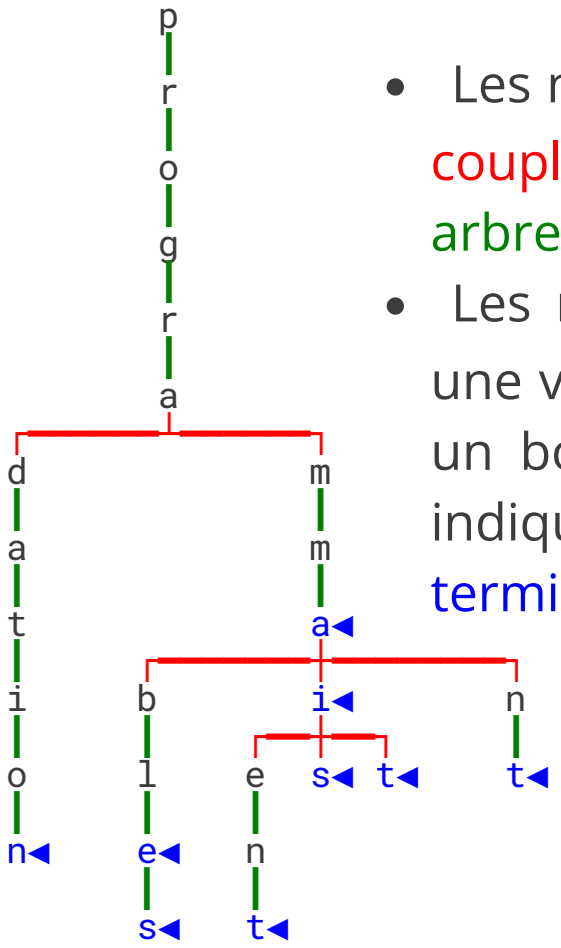
kn@lmf.cnrs.fr
https://usr.lmf.cnrs.fr/~kn

université
PARIS-SACLAY

Résumé de l'épisode précédent

On a vu le *trie*, un structure de donnée permettant d'implémenter des dictionnaires :

	Accès	Données triées	Persistante
`Trie`	O(k)	✔(?)	✔



- Les nœuds sont des listes de couples (caractère, sous-arbre)
- Les nœuds contiennent soit une valeur (dictionnaires) soit un booléen (ensemble) pour indiquer qu'ils sont terminaux.

Plan

Ajout dans un trie (détail)

On revient sur le type des *tries* :

```
type trie = Node of bool * (char * trie) list
```

Remarque 1 : pour simplifier les dessins, on considère la version « ensemble » d'un trie, qu'on généralisera ensuite à la version « dictionnaire »

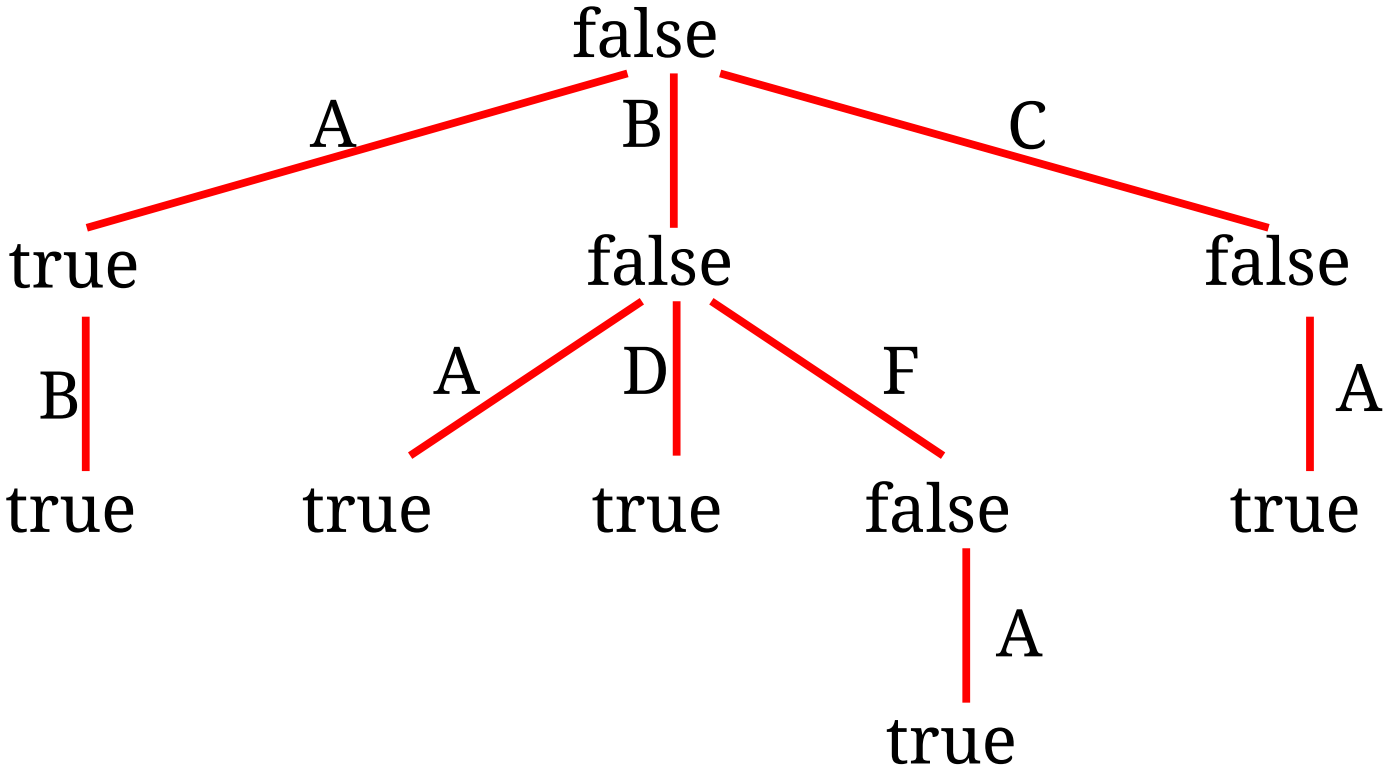
Remarque 2 : pour extraire les composantes d'un type à 1 constructeur, on peut utiliser `let` plutot que `match` :

```
match n with
 Node (b, l) -> ...          →   let Node (b, l) = n in ...
```

Ajout dans un trie (détail)

On donne une présentation (graphique) alternative qui permet de mieux comprendre l'ajout dans un trie

Ajout de `"CA"`



Ajout dans un trie (détail)

```
let add key t =
 let rec insert_node i t =
   let Node (b, l) = t in
     if i = String.length key then Node (true, l)
     else Node (b, insert_list i l)
 and insert_list i l =
   let ci = key.[i] in
   match l with
   | [] -> [ (ci, insert_node (i + 1) empty) ]
   | (d, t) :: ll ->
     if ci > d then (d, t) :: insert_list i ll
     else if ci = d then (ci, insert_node (i + 1) t) :: ll
     else (ci, insert_node (i + 1) empty) :: l
 in
 insert_node 0 t
```

```
add "A" (Node(false, []))
insert_node 0 (Node(false, []))
Node (false, insert_list 0 [])   (* ci == 'A' *)
Node (false,  [('A', insert_node 1 (Node(false, [])))])
Node (false,  [('A', Node(true, []))])
```

Ajout dans un trie (détail)

```
let add key t =
 let rec insert_node i t =
   let Node (b, l) = t in
     if i = String.length key then Node (true, l)
     else Node (b, insert_list i l)
 and insert_list i l =
   let ci = key.[i] in
   match l with
   | [] -> [ (ci, insert_node (i + 1) empty) ]
   | (d, t) :: ll ->
     if ci > d then (d, t) :: insert_list i ll
     else if ci = d then (ci, insert_node (i + 1) t) :: ll
     else (ci, insert_node (i + 1) empty) :: l
 in
 insert_node 0 t
```

```
add "AB" (Node (false,  [('A', Node(true, []))]))
insert_node 0 (Node (false,  [('A', Node(true, []))]))
Node (false, insert_list 0 [('A', Node(true, []))])   (* ci == 'A' *)
Node (false,  ('A', insert_node 1 (Node(true, []))) :: [])
Node (false,  ('A', (Node(true, insert_list 1 []))) :: []) (* ci = 'B' *)
Node (false,  ('A', (Node(true, [ ('B', insert_node 2 (Node(false,[]))))]))) :: [])
Node (false,  ('A', (Node(true, [ ('B', (Node(true,[]))))]))) :: [])
Node (false,  [('A', (Node(true, [ ('B', (Node(true,[]))))]))))])
```

Plan

Trier les mots d'un texte

On considère un texte $T = \{ s_1, ..., s_n \}$ constitué de n mots. Le nombre de caractères dans le texte est

$$|T| = \sum_{i=1..n} |s_i|$$

On considère l'algorithme suivant :

- Créer un *trie* t vide
- Pour i entre 1 et n, insérer s_i dans t
- Parcourir t avec un parcours préfixe pour afficher les clés dans l'ordre

On a donc trié l'ensemble des mots du texte. Quelle est la complexité ?

$$O(|T|)$$ 🤔

On a pu trier un texte en moins que $O(|T|\log(|T|))$, où est l'arnaque ?

Complexité des tris

Il faut être très précis dans les énoncés.

Un tri utilisant une *fonction de comparaison binaire* (qui compare 2 éléments entre eux) doit effectuer $O(N \log(N))$ comparaisons pour une collection de taille N.

Intuition: étant donné une collection de N éléments, il y a $N!$ permutations possibles. En faisant les comparaisons 2 à 2, on peut « trouver » la bonne permutation au mieux en $O(\log(N!)) = O(N \log(N))$

Complexité des tris

Mais ici on n'utilise pas une comparaison 2 à 2. On a une structure de données auxiliaire qui sait donner rapidement un ensemble de clés plus grandes.

On peut donc trier en temps linéaire, mais pour un ordre bien particulier, l'ordre lexicographique. On ne peut pas utiliser un *trie* pour un ordre arbitraire.

Exemple, si on a une liste de vecteurs (x,y) que l'on veut trier par taille croissante ($\sqrt{(x^2+y^2)}$), on ne peut pas utiliser un *trie*.

Ensembles

Dans toute la suite, on suppose que les *tries* qu'on manipule représentent des ensembles et non des dictionnaires. On stocke juste dans les nœuds un booléen qui dit si le nœud est terminal ou pas:

```
type trie =  Node of bool * (char * trie list)
(* pas de 'a, car on ne stocke pas de valeur *)
```

Unicité de la représentation

Pour un ensemble de clés données, le *trie* représentant cet ensemble est unique.

- `{ }` ⇒ `Node(false, [])`
- `{"A"}` ⇒ `Node(false, ['A', Node(true, [])])`
- `{"A", "AA", "BC", "C" }` ⇒

```
Node(false,
[ 'A', Node(true,  ['A', Node(true, [])]);
  'B', Node(false, ['C', Node(true, [])]);
  'C', Node(true,  [])
])
```

Ce n'est pas le cas d'autres structures. Exemple avec les tables de hachage:

- Ajouter "A" dans la table vide
- Ajouter "A" dans la table vide, puis ajouter 10000 autres clés, puis les retirer

Le tableau interne peut avoir été agrandi, mais c'est le même ensemble.

Unicité de la représentation : utilité

Si deux objets (immuables) égaux sont *structurellement égaux*, alors on peut partager leur représentation en mémoire. Cela permet d'accélérer des opérations et d'économiser de la mémoire. Avant de voir cette technique, on doit s'intéresser à la représentation en mémoire des valeurs OCaml.

Plan

Opérations avancées

On propose de coder ensemble quelques opérations sur les ensembles :

```
val iter : (string -> unit) -> trie -> unit
(** itération sur les éléments d'un trie *)

val inter : trie -> trie -> trie
(** intersection de deux trie *)
```