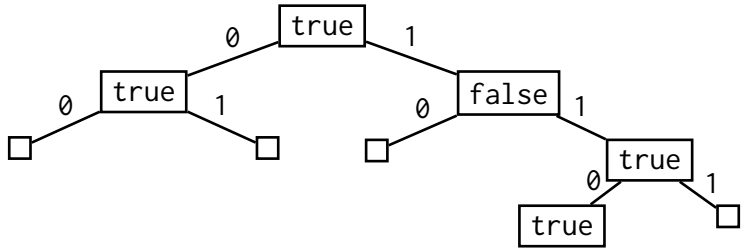
Compilation — Partiel — Automne 2024

Durée 2h. Documents de cours et notes personnelles autorisés. Les exercices sont indépendants.

Exercice 1 (Induction structurelle) La structure de *trie* est un arbre binaire permettant de représenter un ensemble de mots binaires. Pour chaque nœud interne, le fils gauche correspond à la lecture d'un `0` et le fils droit à la lecture d'un `1`. Chaque nœud est associé à un mot : la séquence de `0` et de `1` lue pour atteindre ce nœud en partant de la racine. Chaque nœud interne est également étiqueté par un booléen, indiquant si le mot correspondant est ou non dans l'ensemble représenté. Ainsi, l'arbre ci-dessous représente l'ensemble $\{\varepsilon, \texttt{0}, \texttt{11}, \texttt{110}\}$ (on a omis quelques-uns des sous-arbres vides).



Le type caml suivant permet de représenter un *trie* à l'aide de deux constructeurs : `E` pour un arbre vide et `N` pour un nœud interne, étiqueté par un booléen, avec deux fils.

```
type trie =
  | E
  | N of bool * trie * trie
```

On se donne les équations suivantes pour une fonction in qui indique si une séquence binaire ℓ appartient ou non à un *trie* t :

$$\begin{aligned}
\mathsf{in}(\ell, \mathtt{E}) &= \mathtt{false} \\
\mathsf{in}(\varepsilon, \mathtt{N}(b, t_0, t_1)) &= b \\
\mathsf{in}(\mathtt{0}\ell, \mathtt{N}(b, t_0, t_1)) &= \mathsf{in}(\ell, t_0) \\
\mathsf{in}(\mathtt{1}\ell, \mathtt{N}(b, t_0, t_1)) &= \mathsf{in}(\ell, t_1)
\end{aligned}$$

Questions

1. Écrire le terme représentant le *trie* dessiné en introduction de l'exercice.
2. Définir une fonction size déterminant le cardinal de l'ensemble représenté par un *trie* t.
   Vous pouvez répondre au choix par des équations ou par du code caml.
3. Définir une fonction union qui calcule l'union de deux *tries* t_1 et t_2 donnés en paramètres.
   Vous pouvez répondre au choix par des équations ou par du code caml.
4. Démontrer que pour tous *tries* t_1 et t_2 on a $\mathsf{size}(t_1) \leq \mathsf{size}(\mathsf{union}(t_1, t_2)) \leq \mathsf{size}(t_1) + \mathsf{size}(t_2)$.
5. Démontrer que pour tous *tries* t_1 et t_2 et toute séquence ℓ, si $\mathsf{in}(\ell, t_1)$ alors $\mathsf{in}(\ell, \mathsf{union}(t_1, t_2))$.

□

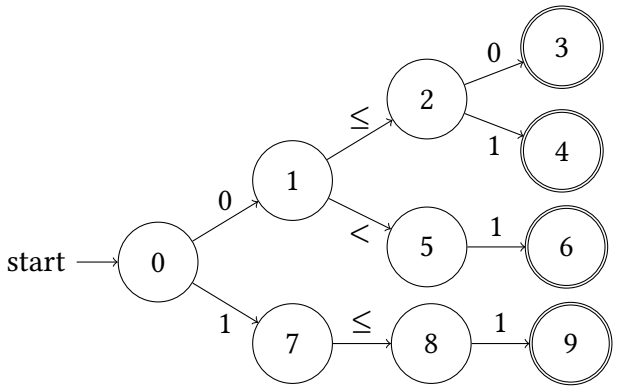
Exercice 2 (Automates) On considère l'alphabet $A = \{a, b, r\}$ et le langage L des mots sur A contenant au moins une occurrence de la séquence « *babar* ».

Questions

1. Donner une expression régulière pour L.
2. Donner un automate déterministe reconnaissant L.

□

Exercice 3 (Langages reconnaissables) On considère l'alphabet $A = \{0, 1, <, \leq\}$. Les mots sur A permettent notamment d'écrire des chaînes de comparaisons de nombres binaires, comme $101 \leq 1000 < 1001 \leq 1001$. L'automate ci-dessous reconnaît le langage L_1 des mots sur A formant une comparaison valide de deux nombres à 1 chiffre.



Questions

1. Donner une expression régulière reconnaissant L_1.
2. Dans l'automate ci-dessus, fusionner les états qui peuvent l'être pour obtenir un automate déteministe reconnaissant L_1 avec aussi peu d'états que possible (on ne veut pas forcément un automate complet).
3. Soit L_2 le langage des chaînes de comparaisons valides de nombres à 1 chiffre. L_2 contient donc par exemple $0 \leq 0 < 1 \leq 1$ ou $0 \leq 1 \leq 1$. Le langage L_2 est-il reconnaissable ? Donner une justification adéquate.
4. Soit L_3 le langage des comparaisons valides de deux nombres de tailles quelconques. L_3 contient donc par exemple $101 < 1001$ mais pas $110 \leq 101$. Le langage L_3 est-il reconnaissable ? Donner une justification adéquate.

□

Exercice 4 (Grammaires) Voici 5 propositions différentes d'une grammaire pour décrire des séquences d'instructions séparées par des points-virgules. Dans tous les cas, on considère un symbole terminal ; (point-virgule), et deux symboles non terminaux *instr* et *seq*, respectivement pour les instructions et pour les séquences. Le symbole ε désigne une séquence vide.

$$G_a \quad \begin{array}{lcl} seq & \to & seq\,;\, seq \\ & | & instr \end{array} \qquad G_b \quad \begin{array}{lcl} seq & \to & seq\; seq \\ & | & instr; \end{array} \qquad G_c \quad \begin{array}{lcl} seq & \to & instr;\, seq \\ & | & \varepsilon \end{array}$$

$$G_d \quad \begin{array}{lcl} seq & \to & seq\,;\, instr \\ & | & \varepsilon \end{array} \qquad G_e \quad \begin{array}{lcl} seq & \to & instr;\, seq \\ & | & instr; \end{array}$$

Questions

1. Parmi les grammaires proposées, lesquelles permettent de dériver la séquence suivante ?

   instr; *instr*; *instr*; *instr*;

   Donner un arbre de dérivation pour chacune.
2. Parmi les grammaires proposées, lesquelles sont ambigues ?
3. Identifier parmi les propositions au moins deux grammaires décrivant exactement le même langage.
4. Pour chaque grammaire qui n'a pas été citée à la question précédente, donner un exemple d'une séquence qui est dérivable avec cette grammaire mais n'est dérivable avec aucune des autres propositions.

□

Exercice 5 (Interprétation) On se donne une syntaxe abstraite minimale pour des expressions arithmétiques avec variables locales, ainsi qu'une fonction d'évaluation écrite en caml.

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Var of string
  | Let of string * expr * expr

let eval e =
  let env = Hashtbl.create 32 in
  let rec eval = function
    | Cst n -> n
    | Add(e1, e2) ->
       let v1 = eval e1 in
       let v2 = eval e2 in
       v1 + v2
    | Var x ->
       let e = Hashtbl.find env x in
       let v = eval e in
       Hashtbl.replace env x (Cst v);
       v
    | Let(x, e1, e2) ->
       Hashtbl.add env x e1;
       let v = eval e2 in
       Hashtbl.remove env x;
       v
  in
  eval e
```

En annexe, un rappel sur le module `Hashtbl` et ses fonctions.

Questions

1. On considère l'expression *e* suivante :

   ```
   let x = 1 + 2 in
   let y = 1 + 4 in
   x + x
   ```

   Détailler l'évaluation de l'expression *e*, en précisant à chaque étape le contenu de la table de hachage. Combien de fois est réalisée l'addition `1+2` ? Combien de fois est réalisée l'addition `1+4` ?
2. À quoi sert la ligne

   ```
   Hashtbl.replace env x (Cst v);
   ```

   du cas `Var` ? Que se passerait-il si on l'enlevait ? L'évaluateur serait-il encore correct ? (justifier ou donner un contre-exemple)
3. À quoi sert la ligne

   ```
   Hashtbl.remove env x;
   ```

   du cas `Let` ? Que se passerait-il si on l'enlevait ? L'évaluateur serait-il encore correct ? (justifier ou donner un contre-exemple)

□

Annexe. Rappel des effets des fonctions sur les tables de hachage (`Hashtbl`) utilisées :

- `Hashtbl.create` *n* crée une nouvelle table de taille *n* (la taille étant susceptible d'être ajustée automatiquement plus tard).
- `Hashtbl.add` *t x v* ajoute une nouvelle association entre la clé *x* et la valeur *v* dans la table *t*, sans détruire les éventuelles autres associations déjà présentes pour *x*.
- `Hashtbl.find` *t x* renvoie la valeur la plus récente associée à *x* dans la table *t*.
- `Hashtbl.remove` *t x* retire de la table *t* l'association la plus récente pour *x*.
- `Hashtbl.replace` *t x v* est équivalent à à la séquence
  `Hashtbl.remove` *t x*; `Hashtbl.add` *t x v*.

Ainsi, si on réalise l'enchaînement

```
let t = Hashtbl.create 32 in
Hashtbl.add t "x" 1;
Hashtbl.add t "x" 2;
Hashtbl.replace t "x" 3;
let a = Hashtbl.find t "x" in
Hashtbl.remove t "x";
let b = Hashtbl.find t "x" in
...
```

on obtiendra pour `a` la valeur 3 et pour `b` la valeur 1.