Introduction
TD 1

■ ■ ■

Exercice 1. (Principe d'induction) En utilisant une mesure appropriée et le principe d'induction sur les entiers, on peut généraliser le principe d'induction à d'autres types :

1. Énoncer et démontrer le principe d'induction pour les listes.

2. Énoncer et démontrer le principe d'induction pour les arbres binaires.

3. Énoncer et démontrer le principe d'induction pour les types inductifs en général.

Exercice 2. (Recherche) Pour chacun des problèmes suivant, on donnera une spécification de ce programme en langage mathématiques et on prouvera sa correction.

1. Un programme qui recherche un entier dans une liste non triée et retourne vrai s'il appartient à la liste, faux sinon.

2. Un programme qui recherche un entier dans un tableau non trié et retourne l'indice de l'entier s'il existe, -1 sinon.

Exercice 3. (Tests et vérification) *extrait du cours et exercices corrigés d'algorithmique de Juliand.*

```
int prog (int x, int [] t) =
    // t est un tableau trié d'entiers contenant une unique occurrence x
    g := 0 ; d := t.length - 1
    while (g <> d) do
         m := (g+d) / 2 ; // division entière
         if (x < t[m]) then d := m-1
         else g := m
       done
       return g ;;
```

1. Simuler l'exécution de ce programme pour $n = 5$, $x = 7$ et t le tableau contenant les éléments 2, 4, 7, 9, 9. Que fait ce programme ?

2. Par quelles méthodes peut-on s'assurer de la fiabilité du programme ? Étudier la réalisabilité de chacune de ces méthodes.

3. Existe-t-il une méthode qui montre que ce programme est incorrect ?

4. Donner une spécification (données, résultats, pré condition, post condition) ?

5. Vérifier la correction du programme.

Exercice 4. (Tri Rapide) L'algorithme dit Quick-sort a été introduit par Hoare en 1961, il repose sur le choix d'un pivot et le partitionnement du tableau par rapport à ce pivot. Il a la forme suivante :

```
tri_rapide(t, premier, dernier) =
  pivot := premier
 if (premier < dernier) then
   pivot := choix_pivot(t,premier,dernier);
   partitionner(t, premier, dernier, pivot);
   tri_rapide(t,premier,pivot-1);
   tri_rapide(t,pivot+1,dernier)
```

1. Donner la spécification d'un algorithme de tri sur un tableau.

2. Donner la spécification, implémenter et prouver les fonctions pivot et partitionner.

3. Prouver l'algorithme de tri Quick-sort.

Logique de Hoare

TD 2

■ ■ ■

On rappelle que les règles d'inférence de la logique de Hoare sont données par :

$$\frac{}{\{P\}\ \texttt{nop}\ \{P\}} \qquad \frac{}{\{P[x := E]\}\ x\ :=\ E\ \{P\}}$$

$$\frac{\{P\}\ C\ \{Q\} \qquad \{Q\}\ D\ \{R\}}{\{P\}\ C; D\ \{R\}}$$

$$\frac{\{E = \texttt{true}\ \wedge\ P\}\ C\ \{Q\} \qquad \{E = \texttt{false}\ \wedge\ P\}\ D\ \{Q\}}{\{P\}\ \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ D\ \{Q\}}$$

$$\frac{\{E = \texttt{true}\ \wedge\ I\ \wedge\ V = z\}\ C\ \{I\ \wedge\ V < z\} \qquad I\ \Rightarrow\ V \geq 0}{\{I\}\ \texttt{while}\ E\ \texttt{do}\ C\ \texttt{done}\ \{E = \texttt{false}\ \wedge\ I\}}$$

$$\frac{P'\ \Rightarrow\ P \qquad \{P\}\ C\ \{Q\} \qquad Q\ \Rightarrow\ Q'}{\{P'\}\ C\ \{Q'\}}$$

(où C, D désignent des commandes, E, V des expressions sans effets de bord).

Exercice 1. Calcul de la racine par addition (*Extrait du livre cours et exercices d'algorithmique de Juliand.)*
On cherche un programme calculant la racine carré entière d'un entier.

1. Donner la spécification du problème.

2. Implémenter une solution naïve du problème qui repose sur une méthode par incrémentation.

3. Annoter et prouver votre implémentation en utilisant la logique de Hoare.
On considère l'amélioration du programme où l'on remplace une multiplication coûteuse par une addition.

```
r := 0; y := 1; z := 1;
while (y<=n) do
  z := z+2; y := y+z; r := r+1;
done
return r;
```

4. À l'aide des axiomes d'affectation, trouver les expressions E_i et S_i afin que les formules suivantes soient valides :

$$\begin{array}{lll}
\{E_1\} & \texttt{z:=1} & \{y = (r+1)^2 \wedge z = 2r+1 \wedge r^2 \leq n\} \\
\{E_2\} & \texttt{y:=1} & \{y = (r+1)^2 \wedge 1 = 2r+1 \wedge r^2 \leq n\} \\
\{E_3\} & \texttt{r:=0} & \{1 = (r+1)^2 \wedge 1 = 2r+1 \wedge r^2 \leq n\} \\
\{E_4\} & \texttt{r:=r+1} & \{y = (r+1)^2 \wedge z = 2r+1 \wedge r^2 \leq n\} \\
\{E_5\} & \texttt{y:=y+z} & \{y = (r+2)^2 \wedge z = 2r+3 \wedge (r+1)^2 \leq n\} \\
\{E_6\} & \texttt{z:=z+2} & \{y+z = (r+2)^2 \wedge z = 2r+3 \wedge (r+1)^2 \leq n\}
\end{array}$$

$$\begin{array}{lll}
\{y = (r+1)^2 \wedge z = 2r+3 \wedge r^2 \leq n\} & \texttt{y:=y+z} & \{S_1\} \\
\{y = (r+1)^2 \wedge z = 2r+1 \wedge r^2 \leq n\} & \texttt{z:=z+2} & \{S_2\}
\end{array}$$

5. À l'aide de la règle de la séquence et de l'affaiblissement, démontrer les théorèmes suivants :

–

$$\{n \geq 0\}$$
$$\texttt{r:=0; y:=1; z:=1}$$
$$\{y = (r+1)^2 \wedge z = 2r+1 \wedge r^2 \leq n\}$$

–

$$\{y <= n \wedge y = (r+1)^2 \wedge z = 2r+1 \wedge r^2 \leq n\}$$
$$\texttt{z:=z+2; y:=y+z; r:=r+1}$$
$$\{y = (r+1)^2 \wedge z = 2r+1 \wedge r^2 \leq n\}$$

6. En appliquant la règle d'itération, montrer la validité de la formule :

$$\{y = (r+1)^2 \wedge z = 2r+1 \wedge r^2 \leq n\}$$
$$\texttt{while (y<=n) do z:=z+2; y:=y+z; r:=r+1 done}$$
$$\{y = (r+1)^2 \wedge z = 2r+1 \wedge r^2 \leq n \wedge y > n\}$$

7. En vous aidant des questions précédentes, annoter le programme et prouver sa correction.

Exercice 2. (Boucle for) En utilisant le codage de `for` à l'aide de `while`, écrire la règle de Hoare correspondant à la construction : `for` i `:=` E `to` F `do` C `done`.

Exercice 3. (Calcul de minimum) On considère le programme suivant, qui calcule le minimum d'une fonction entre 1 et n.

```
m := f(1) ;
for i := 2 to n do
   if f(i) < m then m := f(i)
done
```

1. Quelles sont la précondition et la post-condition de ce programme ? Comment la boucle for doit-elle être annotée ?
2. Calculer les obligations de preuves correspondantes, et vérifier qu'elles sont satisfaites.

Exercice 4. (Tableaux) En logique de Hoare, les tableaux sont manipulés à l'aide des deux fonctions suivantes :
– La fonction `access(`t`,` i`)` qui retourne le i-ème élément du tableau t ;
– La fonction `store(`t`,` i`,` v`)` qui retourne un nouveau tableau ayant les mêmes éléments que t, sauf le i-ème qui est remplacé par v.
Ces deux fonctions permettent de définir la lecture et l'écriture dans un tableau : [1]

$$t[i] \quad \equiv \quad \texttt{access}(t,i) \qquad \text{et} \qquad t[i] := E \quad \equiv \quad t := \texttt{store}(t,i,E)$$

1. Quels axiomes est-il raisonnable de supposer sur les fonctions access et store ?
2. Montrer la correction du programme suivant (où x et y sont des variables fraîches) :

$$\{t[i] = x \wedge t[j] = y\} \;\; v := t[i];\; t[i] := t[j];\; t[j] := v \;\; \{t[i] = y \wedge t[j] = x\}$$

Exercice 5. (Tri sélection) Montrer à l'aide de ce qui précède la correction de l'algorithme de tri suivant :

```
for i := 0 to n - 2 do
   for j := i + 1 to n - 1 do
      if t[j] < t[i] then begin
         tmp := t[i] ;
         t[i] := t[j] ;
         t[j] := tmp
      end
   done
done
```

1. On notera que l'opération qui consiste à écrire dans une seule case du tableau est traduite en logique de Hoare par le remplacement du tableau complet.

Premiers pas en coq

TP 1

■ ■ ■

Pour résoudre les exercices, on s'aidera de la correspondance suivante et de la documentation en ligne :
– le petit guide : http://www.pps.jussieu.fr/~miquel/enseignement/mpri/guide.html
– la documentation complète : http://coq.inria.fr/doc
– un tutoriel et une F.A.Q. sont aussi disponible sur http://coq.inria.fr

Exercice 1. (Calcul propositionnel) Établir en Coq les formules suivantes :

1. `forall A : Prop, A -> A`
2. `forall A B C : Prop, (A -> B) -> (B -> C) -> A -> C`
3. `forall A B : Prop, A /\ B <-> B /\ A`
4. `forall A B : Prop, A \/ B <-> B \/ A`
5. `forall A : Prop, A -> ~~A`
6. `forall A B : Prop, (A -> B) -> ~B -> ~A`
7. `forall A : Prop, ~~(A \/ ~A)`
8. `forall A B C : Prop, (A \/ B) /\ C <-> (A /\ C) \/ (B /\ C)`
9. `forall A B C : Prop, (A /\ B) \/ C <-> (A \/ C) /\ (B \/ C)`

Exercice 2. (Calcul des prédicats) Après avoir effectué les déclarations suivantes

```
Parameter X Y : Set.
Parameter A B : X -> Prop.
Parameter R : X -> Y -> Prop.
```

établir en Coq les formules suivantes :

1. `(forall x : X, A x /\ B x) <-> (forall x : X, A x) /\ (forall x : X, B x)`
2. `(exists x : X, A x \/ B x) <-> (exists x : X, A x) \/ (exists x : X, B x)`
3. `(exists y : Y, forall x : X, R x y) -> (forall x : X, exists y : Y, R x y)`

Que peut-on dire de la réciproque de la dernière formule ?

Exercice 3. (Relations d'ordre) En Coq, on considère un type E : Set muni d'une relation binaire R dont on suppose qu'elle satisfait aux axiomes des relations d'ordre :

```
Parameter E :   Set.
Parameter R :   E -> E -> Prop.

Axiom refl :    forall (x : E),      R x x
Axiom trans :   forall (x y z : E),  R x y -> R y z -> R x z.
Axiom antisym : forall (x y : E),    R x y -> R y x -> x = y.
```

On définit les notions de plus petit élément et d'élément minimal de la façon suivante :

```
Definition smallest (x0 : E) := forall (x : E), R x0 x.
Definition minimal  (x0 : E) := forall (x : E), R x x0 -> x = x0.
```

Quels sont les types des objets `smallest` et `minimal` ?
Énoncer en Coq puis démontrer les lemmes suivants :

1. Si R admet un plus petit élément, alors celui-ci est unique.
2. Le plus petit élément, s'il existe, est un élément minimal.
3. Si R admet un plus petit élément, alors il n'y a pas d'autre élément minimal que celui-ci.

Indications : En Coq, une définition s'utilise en remplaçant le *definiendum* par son *definiens* à l'aide de la tactique `unfold` ⟨*definiendum*⟩ (*unfold* = déplier). L'égalité se traite à l'aide des tactiques `reflexivity`, `symmetry`, `transitivity` ⟨*terme*⟩ et `rewrite` ⟨*hypothèse*⟩.

Exercice 4. (Le paradoxe des buveurs) Dans cet exercice, on suppose la règle de raisonnement par l'absurde, que l'on déclare en Coq de la manière suivante :

```
Axiom not_not_elim : forall A : Prop, ~~A -> A.
```

1. Montrer en Coq que cet axiome entraîne le tiers-exclus : `forall A : Prop, A \/ ~ A`.

On se propose maintenant de formaliser le paradoxe des buveurs, dû à Smullyan :

> *Dans toute pièce non vide on peut trouver une personne ayant la propriété suivante : Si cette personne boit, alors tout le monde dans la pièce boit.*

2. Déclarer en Coq les divers éléments du problème (en s'inspirant de l'exercice 3).
3. Énoncer le paradoxe et en effectuer la preuve (laquelle repose sur le tiers-exclus).

Les entiers naturels en Coq

TP 2

■ ■ ■

En Coq, l'introduction d'un nouveau type de données s'effectue à l'aide d'un mécanisme de *définition inductive* qui ressemble beaucoup à la définition d'un type concret en Caml. Ainsi, le type `nat` des entiers naturels est introduit [1] à l'aide de la définition inductive suivante :

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

Cette définition ajoute à l'environnement courant trois nouvelles constantes :
– le type `nat : Set` (`Set` est le type des petits types) ;
– le constructeur [2] `O : nat` (constructeur constant) ;
– le constructeur `S : nat -> nat` (constructeur à un argument de type `nat`).
Le système utilise ensuite le sucre syntaxique `0`, `1`, `2`, etc. pour désigner les entiers `O`, `S O`, `S (S O)`, `S (S (S O))`, etc.
La définition inductive ci-dessus engendre automatiquement un certain nombre de principes d'induction, dont le plus utilisé en pratique est le schéma de récurrence

```
nat_ind :
  forall P : nat -> Prop,
    P O -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

utilisé en interne par les tactiques `elim` et `induction`.

Exercice 1. – L'addition En Coq, l'addition [3] est définie au moyen de la construction `Fixpoint`, qui est l'équivalent du « `let rec` » de Caml :

```
Fixpoint plus (n m:nat) : nat :=
  match n with
  | O => m
  | S p => S (plus p m)
  end.
```

Il est important de noter que les appels récursifs se font ici sur un premier argument `n` de plus en plus petit. Il s'agit en fait de *décroissance structurelle*. Coq refuse les définitions pour lesquels il n'est pas en mesure de vérifier cette propriété, et qui risquent donc de ne pas forcément terminer [4].
Le système utilise la notation `n + m` pour désigner le terme `plus n m`.

1. Vérifier à l'aide des tactiques `simpl` et `reflexivity` qu'on a les égalités définitionnelles

   `O + m = m` et `S n + m = S (n + m)`.

   A-t-on les égalités définitionnelles `n + O = n` et `n + S m = S (n + m)` ?
2. Montrer les deux lemmes suivants :

   ```
   Lemma plus_n_O  : forall n,    n + 0 = n.
   Lemma plus_n_Sm : forall n m,  n + S m = S (n + m).
   ```

   On prouvera ces deux lemmes par récurrence sur `n`, à l'aide de la tactique `induction`.
3. Montrer que l'addition est commutative : `forall n m, n + m = m + n.`
4. Montrer que l'addition est associative : `forall n m p, (n + m) + p = n + (m + p).`

1. cf fichier `theories/Init/Datatypes.v`
2. Attention ! le constructeur s'appelle `O` (lettre « O ») et non `0` (« zéro »).
3. cf fichier `theories/Init/Peano.v`
4. Dans les versions de Coq antérieures à 8.2, il fallait indiquer l'argument de décroissance à l'aide d'une annotation `{struct n}`.

Exercice 2. – La multiplication En Coq, la multiplication est définie par

```
Fixpoint mult (n m:nat) : nat :=
  match n with
  | O => O
  | S p => m + mult p m
  end.
```

(Le système utilise le sucre syntaxique `n * m` pour désigner le terme `mult n m`.)

1. Vérifier à l'aide des tactiques `simpl` et `reflexivity` qu'on a les égalités définitionnelles

   `O * m = O` et `S n * m = m + n * m.`

2. Montrer la propriété de distributivité : `forall n m p, (n + m) * p = n * p + m * p.`
3. Montrer que la multiplication est commutative et associative.

Exercice 3. – La relation d'ordre On peut définir[5] la relation d'ordre usuelle sur les entiers `le : nat -> nat -> Prop` en posant :

```
Definition le (n m : nat) := exists p, n + p = m.
```

Montrer que `le` est une relation d'ordre :

```
Lemma le_refl :     forall n,      le n n.
Lemma le_trans :    forall n m p,  le n m -> le m p -> le n p.
Lemma le_antisym :  forall n m,    le n m -> le m n -> n = m.
```

Tactiques utiles : `simpl`, `elim`, `induction`, `rewrite`, `generalize`.

5. Cette définition n'est pas celle de la librairie standard de Coq (cf fichier `theories/Init/Peano.v`), mais est bien entendu équivalente.

Un compilateur d'expressions arithmétiques en Coq

TP 3

■ ■ ■

Exercice 1. – Expressions arithmétiques

On considère le type d'expressions arithmétiques défini en Coq par :

```
Inductive expr : Type :=
 | Num : nat → expr            (* Constante entière *)
 | Plus : expr → expr → expr  (* Somme de deux expressions *)
 | Mult : expr → expr → expr. (* Produit de deux expressions *)
```

1. Définir la fonction d'évaluation correspondante `eval : expr → nat`
2. Tester cette fonction à l'aide de la commande `Eval compute in ...`

Exercice 2. – La machine à pile

On considère une machine à pile dont le jeu d'instructions est donné par :

```
Inductive instr : Set :=
 | PUSH : nat → instr  (* Pousser un nombre sur la pile *)
 | ADD : instr         (* Ajouter les deux nombres au sommet de la pile *)
 | MUL : instr.        (* Multiplier les deux nombres au sommet de la pile *)
```

Les notions de programme et de pile sont définis par :

```
Definition prog := list instr.
Definition stack := list nat.
```

On utilisera les listes (polymorphes) de Coq dont les définitions sont chargées à l'aide de la commande `Require Export List`.

1. Définir une fonction `exec_instr : instr → stack → stack` exécutant une instruction dans une pile donnée et retourne l'état de la pile après l'exécution de l'instruction. Quel choix d'implémentation faites-vous dans le cas où la pile ne contient pas assez d'éléments pour exécuter une instruction ? Critiquez ce choix.
2. Définir une fonction `exec_prog : prog → stack → stack` exécutant un programme dans une pile donnée et retourne l'état de la pile après l'exécution de l'instruction.
3. Montrez que la fonction d'exécution de programme est associative :

```
forall (p1 p2 : prog) (s : stack),
 exec_prog (p1 ++ p2) s = exec_prog p2 (exec_prog p1 s).
```

Exercice 3. – Le compilateur

On considère la fonction de compilation définie par :

```
Fixpoint compile (e : expr) : prog :=
  match e with
    | Num n      ⇒ PUSH n :: nil
    | Plus e1 e2 ⇒ compile e2 ++ compile e1 ++ (ADD :: nil)
    | Mult e1 e2 ⇒ compile e2 ++ compile e1 ++ (MUL :: nil)
  end.
```

1. Exprimez en Coq puis démontrez la correction de la fonction de compilation.
2. Quelle est la complexité de la fonction de compilation ?

Une technique de compilation standard [1] consiste à produire le code en sens inverse, à l'aide d'une fonction récursive qui prend en argument l'expression à compiler et le code qui suit :

```
Fixpoint compile_cont (e : expr) (p : prog) : prog :=
  match e with
    | Num n      ⇒ PUSH n :: p
    | Plus e1 e2 ⇒ compile_cont e2 (compile_cont e1 (ADD :: p))
    | Mult e1 e2 ⇒ compile_cont e2 (compile_cont e1 (MUL :: p))
  end.
```

Le programme p est parfois appelé une continuation. De cette fonction on déduit une fonction de compilation à la complexité linéaire :

```
Definition compile_opt (e : expr) := compile_cont e nil.
```

3. Montrez que `compile_opt` produit exactement le même code que compile. En déduire que cette nouvelle fonction de compilation est correcte.

1. C'est cette technique qui est utilisée dans l'implémentation courante du compilateur Caml. En pratique, cette façon de produire le code permet à chaque instant de connaître la suite du code, et d'utiliser cette connaissance pour effectuer des optimisations à la volée.

■ ■ ■

Exercice 1. – Définitions de la clôture réflexive-transitive

Sur un type de données `T : Type` fixé, on cherche à définir la clôture réflexive-transitive d'une relation `R : T → T → Prop`, qui est par définition la plus petite relation réflexive et transitive contenant la relation `R`. Il est naturel d'introduire en Coq cette notion au moyen de la définition inductive suivante (paramétrée par la relation `R`) :

```
Inductive clos1 (R : T → T → Prop) : T → T → Prop :=
  | cl1_base : forall x y, R x y → clos1 R x y
  | cl1_refl : forall x, clos1 R x x
  | cl1_trans : forall x y z, clos1 R x y → clos1 R y z → clos1 R x z.
```

Cependant, il est souvent commode dans les démonstrations de définir la notion de clôture réflexive-transitive d'une manière un peu différente, à savoir comme la relation `R'` engendrée par les règles suivantes :

1. Pour tout `x, R' x x` (cas de base) ;
2. Si `R' x y` et `R y z`, alors `R' x z` (cas inductif).

Cette définition alternative se modélise en Coq au moyen de la définition inductive suivante :

```
Inductive clos2 (R : T → T → Prop) : T → T → Prop :=
  | cl2_refl : forall x, clos2 R x x
  | cl2_next : forall x y z, clos2 R x y → R y z → clos2 R x z.
```

Le but de cet exercice est de montrer l'équivalence des deux définitions. Pour ce faire, on pourra suivre les étapes suivantes :

1. Montrer que `clos2 R x y` entraîne `clos1 R x y` (pour tous `x, y : T`).
2. Montrer que `R x y` entraîne `clos2 R x y` (pour tous `x, y : T`).
3. Montrer que `clos2 R` est une relation transitive.
4. En déduire que `clos1 R x y` entraîne `clos2 R x y` (pour tous `x, y : T`).
5. Montrer que l'opération de clôture réflexive-transitive est idempotente, c'est-à-dire que : `clos1 (clos1 R) x y` ⇔ `clos1 R x y` pour tous `x, y : T`.

Exercice 2. – Définitions de la clôture réflexive-transitive (suite)

On travaille toujours avec un type `T : Type` fixé.

1. Définir la relation identité `id : T → T → Prop` ainsi que l'opérateur de composition de relations `comp : (T → T → Prop) → (T → T → Prop) → T → T → Prop`.
2. Définir une fonction `puiss : (T → T → Prop) → nat → T → T → Prop` telle que `puiss R n` calcule la puissance n-ième de la relation (par l'opération de composition).

Une troisième définition de la clôture réflexive-transitive d'une relation `R` est donnée par la réunion des puissances successives de `R`, c'est-à-dire par :

```
Definition clos3 (R : T → T → Prop) (x y : T) := exists n, puiss R n x y.
```

3. Montrer que cette définition est équivalente aux deux précédentes (`clos1` et `clos2`).

■ ■ ■

On considère un développement en Coq paramétré par un type de données `A : Type`.

Exercice 1. – Arbres binaires

Le type tree des arbres binaires dont les nœuds sont étiquetés par des éléments de type A est défini en Coq par :

```
Inductive tree : Type :=
 | Leaf : tree
 | Node : A → tree → tree → tree.
```

On utilisera les bibliothèques suivantes :

```
Require Import Arith Max Omega.
```

1. Observer la définition de la fonction `max : nat → nat → nat` qui retourne le plus grand de ses deux arguments. Énoncer et rechercher ses propriétésdéfinies par des Lemmes dans la bibliothèque Max.
2. Définir en Coq des fonctions `height` et `size` de type `tree → nat` qui calculent respectivement la hauteur et le nombre de nœuds d'un arbre. (On supposera que `height Leaf = size Leaf = 0`.)
3. Établir que pour tout arbre `t` on a `height t <= size t`. Sur quel lemme purement arithmétique repose l'argument central de cette preuve ?
4. On s'intéresse à un prédicat `elt : A → tree → Prop` exprimant que son premier argument appartient à l'arbre donné en second argument. Donner deux définitions de ce prédicat, l'une inductive, l'autre par point-fixe, et prouver leur équivalence.

On munit maintenant `A` d'une relation d'ordre total `less`, introduite à l'aide des déclarations suivantes :

```
Parameter less : A → A → Prop.
Notation "x <= y" := (less x y).
Axiom less_refl  : forall x : A,      x <= x.
Axiom less_trans : forall x y z : A,  x <= y → y <= z → x <= z.
Axiom less_asym  : forall x y : A,    x <= y → y <= x → x = y.
Axiom less_total : forall x y : A,    x <= y ∨ y <= x.
```

Exercice 2. – Arbres de recherche

On dit qu'un arbre `t : tree` est un arbre binaire de recherche (a.b.r.) si dans tout sous-arbre de `t` de la forme `Node x t1 t2`, tous les éléments de `t1` sont inférieurs ou égaux à `x` tandis que tous les éléments de `t2` sont supérieurs ou égaux à `x`.

1. Définir en Coq des relations `tree_le` et `tree_ge` de type `tree → A → Prop` qui expriment que tous les éléments de l'arbre donné en premier argument sont inférieurs ou égaux (resp. supérieurs ou égaux) à la valeur donnée en second argument.
2. Donner une définition inductive en Coq du prédicat `abr : tree → Prop` qui exprime que son argument est un a.b.r.

On suppose que la comparaison entre éléments de type `A` est effectuée au moyen d'une fonction `cmp` de type `A → A → bool` déclarée en Coq par :

```
Parameter cmp : A → A → bool.
Axiom cmp_less : forall x y : A, cmp x y = true ↔ x <= y.
```

Exercice 3. – Insertion dans un a.b.r.

1. À l'aide de la fonction de comparaison introduite ci-dessus, définir en Coq une fonction d'insertion `insert : A → tree → tree` qui insère un élément `x : A` dans un arbre `t : tree` tout en préservant la structure d'a.b.r. lorsque `t` est déjà un a.b.r.
2. Comment s'énonce la correction de la fonction d'insertion
   – vis-à-vis de la relation d'appartenance `elt` ?
   – vis-à-vis de la structure d'a.b.r. ?
   Énoncer ces deux invariants de correction en Coq.
3. Sans faire la preuve, dire sur quel(s) type(s) d'induction reposent les preuves des deux invariants de correction énoncés ci-dessus. L'hypothèse `less_total` (qui exprime que la relation d'ordre `less` est totale) intervient-elle dans ces deux preuves ? Si oui, où ? On illustrera chaque problème recensé par un contre-exemple bien choisi.
4. Prouver en Coq les deux invariants de correction.