Programmation Avancée (INF441)
Contrôle classant
CORRIGÉ

7 juin 2016

Solution de la question 1 La solution la plus simple et la plus élégante est donnée dans la figure 1.

Cette solution a un défaut : comme les compilateurs Java n'optimisent pas les appels terminaux, l'écriture récursive de `contains` exige un espace $O(n)$ sur la pile, où n est la longueur de la liste. Pour éviter cela, on pouvait écrire une boucle. Cette solution, proposée par certains élèves, a reçu la note maximale, si la boucle était correctement écrite.

On pouvait supposer (même si le sujet ne le précisait pas explicitement) que les éléments de la liste ne sont pas `null`. Toute comparaison avec `null` était donc inutile. □

Solution de la question 2 Les méthodes `isEmpty`, `head`, `tail` ont une complexité $O(1)$. La méthode `contains` exige, dans le pire cas (qui a lieu lorsque l'élément recherché n'apparaît pas dans la liste), un parcours de toute la liste `this` et à chaque étape une comparaison, c'est-à-dire un appel à `equals`. Sa complexité est donc $O(cn)$, où c est le coût d'une comparaison et n est la longueur de la liste. Cela se simplifie en $O(n)$ si l'on suppose que le coût d'une comparaison est constant, c'est-à-dire si l'on suppose que c est $O(1)$.

Notons que les méthodes `hashCode` et `equals` implémentées dans la classe `Object` sont de complexité $O(1)$. Cependant, les méthodes `hashCode` et `equals` et de la classe `E` ne sont pas forcément les mêmes, et peuvent avoir une complexité plus importante. □

Solution de la question 3 Une solution apparaît dans la figure 2. Il faut d'abord calculer le code de hachage de l'élément `e`, à savoir `e.hashCode()`. Ce code est un entier arbitraire. Il faut alors le normaliser de façon à obtenir un indice valide, c'est-à-dire un entier compris entre `0` au sens large et `table.length` au sens strict. (Notons que, au lieu de `table.length`, on peut aussi écrire `N`.)

La solution `e.hashCode() % table.length` est **incorrecte**, car l'opérateur `%` de Java produit un résultat négatif lorsque ses arguments ne sont pas de même signe.

La solution `Math.abs(e.hashCode()) % table.length` est en principe incorrecte également, parce que le résultat de `Math.abs` n'est pas toujours positif : `Math.abs(Integer.MIN_VALUE)` est égal à `Integer.MIN_VALUE`. (Oui, c'est étrange.) Dans le cadre du contrôle, cette solution reçoit la note maximale. La solution `Math.abs(e.hashCode() % table.length)` est correcte. On pouvait aussi calculer `e.hashCode() % table.length`, tester s'il est strictement négatif, et si oui, lui ajouter `table.length`.

La solution proposée ici utilise la méthode `Math.floorMod(x, y)`, dont le résultat appartient à l'intervalle semi-ouvert `[0, y)` lorsque le second argument `y` est positif. D'autres solutions encore étaient possibles, par exemple `(e.hashCode() & Integer.MAX_VALUE) % table.length`. □

Solution de la question 4 Notons que le reste de la division euclidienne est calculé à l'aide d'une séquence fixe d'instructions de la machine, et non pas par soustraction itérée ! Sa complexité est donc $O(1)$. La complexité en temps de la méthode `index` est alors dominée par l'appel à `hashCode`. Elle est donc $O(h)$, où h représente le coût d'un appel à `hashCode`. Le plus souvent, la méthode `hashCode` est écrite de telle manière que h est $O(1)$. □

Solution de la question 5 Une solution est donnée dans la figure 2. L'appel `index(e)` donne l'indice `i` où l'élément `e` doit être stocké s'il appartient à l'ensemble. Il reste à déterminer si `e` apparaît ou non dans la liste `table[i]`, ce que permet l'appel `table[i].contains(e)`. □

Solution de la question 6 D'après les solutions des questions 2 et 4, la complexité de la méthode `contains` est $O(h + cn)$, où n est la longueur maximale des listes `table[i]`. Dans le cas le pire, les collisions peuvent être nombreuses (c'est le cas, par exemple, si la fonction de hachage est constante) : l'entier n n'est alors borné que par le nombre d'éléments de l'ensemble. Toutefois, en pratique, si la fonction de hachage est bien choisie, on peut espérer que les éléments soient répartis de façon approximativement uniforme dans les différentes cases du tableau `table` : l'entier n sera alors approximativement égal au quotient du nombre d'éléments de l'ensemble par la taille du tableau. □

Solution de la question 7 Une solution est donnée dans la figure 2. Comme dans `contains`, on détermine d'abord si `e` appartient déjà à l'ensemble. (On pourrait faire cela en appelant `contains`, mais cela nous conduirait à devoir appeler deux fois `e.hashCode()` dans le cas où l'élément n'appartient pas déjà à l'ensemble.) Si oui, il faut renvoyer **`true`**. Si non, on ajoute l'élément à l'ensemble, en allouant une nouvelle cellule `Cons` en tête de la liste `table[i]`, puis on renvoie **`false`**.

Notons que la bibliothèque `HashSet` de Java adopte la convention inverse : la méthode `add` renvoie **`true`** si l'élément a effectivement été ajouté, donc était absent. Ce piège n'était pas intentionnel !

La formulation de l'énoncé a amené certains élèves à se demander s'il fallait que l'élément figure deux fois dans la table lorsqu'il est ajouté deux fois. L'énoncé indiquait qu'un objet de type `HashSet<E>` représente un ensemble d'éléments, et non pas un multi-ensemble, donc il fallait que chaque élément figure une fois dans l'ensemble. Si on autorisait les doublons, alors la méthode `contains` n'aurait plus la complexité espérée, et (pire) l'itérateur présenterait des doublons à l'utilisateur. Donc, en principe, il ne fallait pas autoriser les doublons. Toutefois, dans le cadre du contrôle, les (nombreuses) copies qui autorisaient les doublons n'ont pas été pénalisées. □

Solution de la question 8 La complexité de `add` est la même que celle de `contains`, à savoir $O(h + cn)$, où n est la longueur maximale des listes `table[i]`. □

Solution de la question 9 Une solution est donnée dans la figure 2. On utilise la syntaxe des classes anonymes pour construire et renvoyer un nouvel itérateur. On munit cet objet de deux champs modifiables, `i` et `rest`. Ces deux champs satisfont les invariants suivants :

1. L'entier `i`, toujours compris entre `0` au sens strict et `table.length` au sens large, va croissant. Les listes `table[j]`, où `j` est strictement inférieur à `i-1`, ont déjà été entièrement traitées : leurs éléments ont été soumis au client. La liste `table[i-1]` a été partiellement traitée : une partie (préfixe) de ses éléments ont été soumis au client. Les listes `table[j]`, où `j` est supérieur ou égal à `i`, n'ont pas été traitées : leurs éléments n'ont pas été soumis au client.
2. La liste `rest` est un suffixe de la liste `table[i-1]`. Elle contient les éléments de la liste `table[i-1]` qui n'ont pas encore été soumis au client.

Notre méthode `hasNext` a pour double effet de déterminer s'il reste ou non des éléments non encore soumis au client et de mettre à jour, si besoin, les champs `i` et `rest`. Cette méthode s'écrit ainsi :

1. Si `rest` est non vide, c'est facile, il reste au moins un élément non encore soumis au client.
2. Dans le cas contraire, on doit chercher plus loin dans le tableau, si cela est possible, c'est-à-dire si `i` est strictement inférieur à `N`. On ré-initialise `rest` à la valeur `table[i]`, puis on incrémente `i`, et on continue la recherche, ce que permet la boucle **`while`**. (Au lieu d'une boucle, on pourrait employer un appel récursif à `hasNext`. Malheureusement, les compilateurs Java n'optimisent pas les appels terminaux, donc une formulation itérative est préférable ici.)
3. Si `rest` est vide et si `i` est égal à `table.length`, alors tous les éléments ont été soumis au client. On renvoie **`false`**.

La méthode `next` s'écrit facilement à l'aide de `hasNext`. Si l'appel `hasNext()` renvoie **`false`**, il faut lancer une exception. S'il renvoie **`true`**, alors il modifie au passage les champs `i` et `rest`, de telle

manière que, à l'issue de l'appel, la liste `rest` est nécessairement non vide. (Il suffit de relire le code de `hasNext` pour s'en convaincre.) Il ne reste donc plus qu'à renvoyer le premier élément de la liste `rest` et à avancer d'un cran le pointeur `rest` le long de cette liste, de façon à ce que le prochain appel à `next` renvoie bien l'élément suivant, et non pas le même élément.

Certains élèves ont proposé un itérateur qui supprime les éléments de la table au fur et à mesure qu'il avance. Cela n'était pas permis. Certains ont proposé de contourner ce problème en créant initialement une copie de la table : cela donnait une solution correcte mais inefficace, puisqu'elle exige un espace auxiliaire de la même taille que la table elle-même.

Signalons que la mise à jour du pointeur `rest` doit se faire dans la méthode `next`, et non pas dans `hasNext`. En effet, la méthode `hasNext` ne doit pas faire avancer l'itérateur ! Deux appels successifs à `hasNext` doivent avoir le même effet qu'un seul appel à `hasNext`. □

Solution de la question 10 La méthode `iterator` alloue, initialise, et renvoie un objet ; rien de plus. Sa complexité en temps est donc $O(1)$.

Considérons maintenant un ensemble contenant n éléments, et notons N la taille du tableau `table`. La complexité en temps, dans le cas le pire, de la méthode `next` est dominée par l'appel à `hasNext`, puisqu'en dehors de cet appel, `next` n'effectue que des opérations de coût $O(1)$. La complexité en temps, dans le cas le pire, de la méthode `hasNext` n'est pas $O(1)$. En effet, à cause de la boucle `while`, dans le cas le pire (où le tableau ne contient que des listes vides), le coût d'un appel à `hasNext` est proportionnel à N. Fort heureusement, comme l'indice `i` ne peut que croître, le coût du parcours du tableau est amorti (réparti) sur les appels successifs à `hasNext`. Plus précisément, le coût de n (ou $2n$) appels successifs à `hasNext` est non pas $O(nN)$, comme on pourrait le craindre, mais $O(n + N)$. Telle est donc la complexité en temps d'une énumération complète.

Enfin, la complexité en espace d'une énumération est $O(1)$. En effet, elle n'exige qu'un seul objet dans le tas, à savoir l'itérateur lui-même. (La sous-liste dont l'adresse est stockée dans le champ `rest` fait partie de la représentation en mémoire de l'ensemble. Elle n'est pas allouée pour les besoins de l'énumération.) De plus, les méthodes `hasNext` et `next` n'étant pas récursives, leurs appels n'exigent qu'un espace $O(1)$ sur la pile. □

Solution de la question 11 Une solution est donnée dans la figure 3. Il suffit d'énumérer les éléments de l'ensemble `set2`, via une boucle « **foreach** », et de les ajouter successivement à l'ensemble `set1`.

Pour que ce code ait un sens, il n'est pas nécessaire que `set2` soit de type `HashSet<E>`. Il suffit d'exiger que `set2` soit de type `Iterable<E>`. Grâce à cette modification, le deuxième argument de la méthode `add` peut être une collection de nature quelconque.

De même, pour que ce code ait un sens, il n'est pas nécessaire que `set1` soit de type `HashSet<E>`. Il suffirait d'exiger que `set1` ait une méthode `boolean add (E e)`. On pourrait faire cela en définissant une interface `Add<E>` qui exige uniquement l'existence de cette méthode, en déclarant que la classe `HashSet<E>` implémente l'interface `Add<E>`, et en donnant à `set1` le type `Add<E>`. □

Solution de la question 12 La table de vérité d'une formule F est un tableau de 2^n booléens. Sa construction exige un temps proportionnel à $2^n . |F|$, où $|F|$ est la taille de la formule F. Cet algorithme n'est donc pas viable : son coût en espace et en temps est trop important. On peut réduire la complexité en espace à $O(1)$ en notant qu'il n'est pas nécessaire de construire en mémoire les deux tables de vérité : il suffit de calculer et de comparer un seul élément à la fois. Toutefois, la complexité en temps restera exponentielle dans le pire cas où les deux formules sont équivalentes.

Notons que le problème de déterminer si deux formules F et F' sont équivalentes est NP-complet. En effet, on peut ramener le problème SAT à ce problème : pour déterminer si une formule F est satisfiable, il suffit de demander si $\neg F$ est invalide, c'est-à-dire non équivalente à *vrai*.

L'approche étudiée dans la suite, à savoir la construction de diagrammes binaires de décision, a également un coût exponentiel (en temps et en espace) dans le pire cas, mais se comporte souvent beaucoup mieux en pratique. □

Solution de la question 13 La solution est donnée dans la figure 4. □

Solution de la question 14 La solution est donnée dans la figure 4. Pour représenter la formule x_i, on utilise la formule « *si* x_i *alors vrai sinon faux* », qui est équivalente. De plus, cette formule est bien réduite (puisque ses sous-arbres sont distincts) et ordonnée (puisque ses sous-arbres ne mentionnent aucune variable). Notons que cette solution est unique : il n'y a pas d'autre arbre réduit, ordonné, et équivalent à la formule x_i. □

Solution de la question 15 La solution est donnée dans la figure 4. Aux feuilles `C`, on utilise la négation Booléenne, `not`. Aux nœuds `D`, il suffit d'appliquer l'opération `neg` récursivement aux deux sous-arbres, puisque la formule $\neg$(*si* x_i *alors T sinon F*) est équivalente à *si* x_i *alors* $\neg T$ *sinon* $\neg F$. On se convainc facilement que, si l'arbre `b` est réduit et ordonné, alors l'arbre `neg b` l'est également. □

Solution de la question 16 La solution est donnée dans la figure 4. Si la condition `t` $\neq$ `f` est satisfaite, alors il est permis de construire l'arbre `D(i, t, f)`, car cet arbre est réduit. On renvoie donc cet arbre. Si au contraire les arbres `t` et `f` sont identiques, alors la formule « *si* x_i *alors T sinon F* » est équivalente à *T*. On renvoie donc l'arbre `t`. (On pourrait aussi bien renvoyer `f`.) Pour déterminer si les arbres `t` et `f` sont identiques, on peut employer l'égalité d'OCaml, notée `=`. Cette fonction polymorphe, de type `'a -> 'a -> bool`, compare deux arbres en profondeur. Si on préférait éviter de l'utiliser, on pourrait définir une fonction récursive `equal` de type `bdt -> bdt -> bool`. □

Solution de la question 17 La solution est donnée dans la figure 4. Il faut analyser les deux arbres simultanément. Si l'un des deux est une feuille `C`, le résultat est immédiat, puisque *vrai* $\wedge$ *F* est équivalente à *F* et *faux* $\wedge$ *F* est équivalente à *faux*. Reste le cas où l'on a affaire à deux nœuds `D`.

Si les deux nœuds portent sur la même variable, alors on exploite le fait que la formule :

$$(\textit{si } x_i \textit{ alors } T_1 \textit{ sinon } F_1) \wedge (\textit{si } x_i \textit{ alors } T_2 \textit{ sinon } F_2)$$

est équivalente à :

$$\textit{si } x_i \textit{ alors } (T_1 \wedge T_2) \textit{ sinon } (F_1 \wedge F_2)$$

Pour construire ce nouvel arbre, on doit utiliser la fonction auxiliaire `d`, et non pas directement le constructeur `D`, car les formules $T_1 \wedge T_2$ et $F_1 \wedge F_2$ pourraient être équivalentes. En utilisant `D`, on risquerait de construire un arbre non réduit. Les conditions `i < t1`, `i < f1`, `i < t2`, `i < f2`, sont nécessairement satisfaites, parce que les arbres `b1` et `b2` étaient ordonnés. Le nouvel arbre est donc lui aussi ordonné.

Si les deux nœuds portent sur des variables distinctes, alors on peut exploiter le fait que la formule :

$$(\textit{si } x_{i1} \textit{ alors } T_1 \textit{ sinon } F_1) \wedge B_2$$

est équivalente à :

$$\textit{si } x_{i1} \textit{ alors } (T_1 \wedge B_2) \textit{ sinon } (T_2 \wedge B_2)$$

À nouveau, le fait d'utiliser la fonction auxiliaire `d` garantit que le nouvel arbre est réduit. Pour qu'il soit ordonné, il faut qu'on ait `i1 < b2`, c'est-à-dire (`b2` étant lui-même ordonné) `i1 < i2`. On utilise donc la construction ci-dessus quand la condition `i1 < i2` est satisfaite, et la construction symétrique quand la condition `i2 < i1` est satisfaite. □

Solution de la question 18 Il ne faut pas donner accès à la fonction `d`, car celle-ci permet de construire des arbres non ordonnés : il suffit pour cela d'appeler `d i t f` en violant l'une des conditions `i < t` ou `i < f`. Les constantes `zero` et `one` et les fonctions `var`, `neg`, et `conj` au contraire sont sans danger : si leurs arguments sont des arbres réduits et ordonnés, elle construisent un arbre réduit et ordonné. Si on n'exporte qu'elles, seuls des arbres qui respectent ces deux invariants pourront être construits. Notons, de plus, qu'elles suffisent à construire toutes les formules logiques possibles. □

Solution de la question 19 Si on ne fournissait à l'utilisateur que les constantes `zero` et `one` et les fonctions `var`, `neg`, et `conj`, alors il pourrait construire toutes les formules logiques possibles, mais ne pourrait rien en faire d'utile, puisqu'aucune de ces fonctions ne permet d'inspecter une formule.

Le problème qui nous motivait initialement était de déterminer si deux formules F et F' sont ou non équivalentes. Pour résoudre ce problème, il faut permettre à l'utilisateur d'effectuer ce test. Comme les arbres binaires de décision réduits et ordonnés sont des formes canoniques, pour déterminer si deux formules sont équivalentes, il suffit de tester si les deux arbres qui les décrivent sont identiques. Pour cela, il suffit de donner accès à un test d'égalité de type `bdt -> bdt -> bool`. À nouveau, pour implémenter ce test, on peut employer soit l'opération `=` d'OCaml, soit une fonction récursive.

On pourrait également donner accès à diverses fonctions utilitaires, comme des fonctions d'affichage. □

Solution de la question 20 Une solution est donnée dans la figure 5.

Pour déterminer si deux nœuds `bdd1` et `bdd2` représentent la même formule, on pourrait écrire une fonction récursive qui compare en profondeur ces nœuds et leurs descendants. Ou bien, on pourrait écrire `bdd1 = bdd2`, qui emploie la fonction primitive d'égalité polymorphe d'OCaml, laquelle effectue également un parcours récursif. On pourrait aussi écrire `bdd1.nature = bdd2.nature`, qui utilise la fonction primitive d'égalité polymorphe et compare seulement les champs `nature`. On pourrait encore écrire `bdd1.identity = bdd2.identity`, qui ne compare que deux entiers. Enfin, on pourrait écrire `bdd1 == bdd2`, qui compare les adresses des objets `bdd1` et `bdd2`. En fait, grâce au partage maximal, toutes ces écritures sont équivalentes ! On utilise donc la comparaison la moins coûteuse, à savoir `bdd1 == bdd2`, qui exige seulement une comparaison (et aucune lecture en mémoire). Son coût est $O(1)$.

La fonction `hash` doit associer à chaque nœud un entier quelconque, et doit si possible n'avoir pas de collisions, c'est-à-dire associer à deux nœuds distincts deux entiers distincts. Ici, c'est possible : il suffit pour cela de renvoyer le contenu du champ `identity`. □

Solution de la question 21 Une solution est donnée dans la figure 5.

La fonction `nature_equal` compare les étiquettes de ses arguments, puis (si les étiquettes sont les mêmes) compare leurs champs. Les comparaisons `equal t1 t2` et `equal f1 f2` se font à l'aide de la fonction `equal` de la question précédente.

La fonction `nature_hash` convertit d'abord chaque champ en un entier. Dans les cas des champs `t` et `f`, cela se fait à l'aide de la fonction `hash` de la question précédente. Puis, si nécessaire, elle combine les entiers ainsi obtenus, à l'aide de la fonction `combine`, pour obtenir un seul entier. La fonction `nature_hash` n'est pas injective, mais cela n'est pas nécessaire. □

Solution de la question 22 La solution est donnée dans la figure 5. □

Solution de la question 23 L'énoncé aurait pu préciser que le paramètre entier de la fonction `create` est une indication de taille, mais n'a pas d'importance fondamentale ; et que la fonction `find` lance l'exception `Not_found` si l'élément n'appartient pas à la table.

Une solution est donnée dans la figure 6. Lorsque l'appel à `memo` a lieu, on crée une table de hachage qui à des clefs de type `nature` associe des données de type `'a`. (Son type est `'a N.t`.) Cette table est initialement vide. Ensuite, on renvoie une fonction anonyme `fun x -> ...` qui correspond à la fonction nommée `f'` dans l'énoncé. À chaque fois qu'on lui soumet un élément `x`, elle consulte la table, via un appel à `N.find`. Si `x` n'est pas déjà connu, elle appelle `f x` pour obtenir une valeur `y`, puis ajoute à la table une association de `x` à `y`, avant de renvoyer `y`. Si `x` est déjà connu, elle trouve dans la table la valeur `y` correspondante, qu'elle renvoie. □

Solution de la question 24 Une solution est donnée dans la figure 6. Afin de garantir que chaque nouveau nœud reçoit une identité unique, il faut utiliser un compteur ; on commence donc par allouer un compteur, `next`, qui à chaque appel produira une nouvel entier. La fonction `fun nature ->`

`{ identity = next(); nature = nature }`, dont le type est `nature -> bdd`, répond alors à la question, à ceci près qu'elle ne respecte pas le partage maximal : si on l'applique deux fois au même argument, elle produit deux nœuds de même nature, mais d'adresses et d'identités distinctes. Pour que le partage maximal soit respecté, il suffit de mémoiser cette fonction. Ainsi, un nouveau nœud ne sera alloué que dans le cas où il n'existe pas déjà un nœud de même nature. □

Solution de la question 25 La solution est donnée dans la figure 6. Elle est la même que dans les des **arbres** binaires de décision, à deux détails près. D'abord, il faut utiliser `construct` à chaque que l'on veut construire un nœud. Ensuite, dans la fonction `d`, le test d'égalité entre les formules `t` et `f` se fait maintenant à l'aide de la fonction `equal`, donc en temps $O(1)$. □

```
class Nil<E> extends List<E> {
  boolean isEmpty () {
    return true;
  }
  E head () throws NoSuchElementException {
    throw new NoSuchElementException ();
  }
  List<E> tail () throws NoSuchElementException {
    throw new NoSuchElementException ();
  }
  boolean contains (E e) {
    return false;
  }
}

class Cons<E> extends List<E> {
  final E       head;
  final List<E> tail;
  Cons (E head, List<E> tail) {
    this.head = head;
    this.tail = tail;
  }
  boolean isEmpty () {
    return false;
  }
  E head () {
    return this.head;
  }
  List<E> tail () {
    return this.tail;
  }
  boolean contains (E e) {
    return head.equals(e) || tail.contains(e);
  }
}
```

FIGURE 1 – Les classes `Nil<E>` et `Cons<E>`

```
public class HashSet<E> implements Iterable<E> {
  private final int N = 1024;
  private final List<E>[] table;
  @SuppressWarnings({"unchecked"}) public HashSet () {
    table = (List<E>[]) new List [N];
    for (int i = 0; i < table.length; i++)
      table[i] = new Nil<E> ();
  }
  private int index (E e) {
    return Math.floorMod(e.hashCode(), table.length);
  }
  public boolean contains (E e) {
    int i = index(e);
    return table[i].contains(e);
  }
  public boolean add (E e) {
    int i = index(e);
    if (table[i].contains(e))
      return true;
    table[i] = new Cons<E> (e, table[i]);
    return false;
  }
  public Iterator<E> iterator () {
    return new Iterator<E> () {
      private int i = 1;                // 0 < i <= table.length
      private List<E> rest = table[0]; // a suffix of table[i-1]
      public boolean hasNext () {
        while (true) {
          if (!rest.isEmpty())          // If rest is nonempty,
            return true;                // then all is well;
          if (i < table.length) {       // otherwise try getting
            rest = table[i];            // new elements from table[i]
            i++;                        // and advancing i.
          }
          else                          // If i cannot be advanced,
            return false;               // then we are finished.
        }
      }
      public E next () throws NoSuchElementException {
        if (!hasNext())                 // This advances i if needed.
          throw new NoSuchElementException ();
        E e = rest.head();              // rest is now nonempty!
        rest = rest.tail();             // Advance the iterator.
        return e;
      }
    };
  }
}
```

FIGURE 2 – La classe HashSet<E>

```
  static <E> void add (HashSet<E> set1, Iterable<E> set2) {
    for (E e : set2)
      set1.add(e);
  }
```

FIGURE 3 – La méthode statique add

```
type variable = int
type bdt =
  | C of bool
  | D of variable * bdt * bdt

let zero = C false
let one  = C true

let var i =
  D (i, one, zero)

let d i t f =
  if t = f then t else D (i, t, f)

let rec neg b =
  match b with
  | C c ->
      C (not c)
  | D (i, t, f) ->
      D (i, neg t, neg f)

let rec conj b1 b2 =
  match b1, b2 with
  | C true, _ ->
      b2
  | _, C true ->
      b1
  | C false, _
  | _, C false ->
      zero
  | D (i1, t1, f1), D (i2, t2, f2) ->
      if i1 = i2 then
        d i1 (conj t1 t2) (conj f1 f2)
      else if i1 < i2 then
        d i1 (conj t1 b2) (conj f1 b2)
      else
        d i2 (conj b1 t2) (conj b1 f2)
```

FIGURE 4 – Arbres binaires de décision

```
type variable = int
type identity = int
type bdd = {
    identity: identity;
    nature: nature;
  }
and nature =
  | C of bool
  | D of variable * bdd * bdd

let equal bdd1 bdd2 =
  bdd1 == bdd2

let hash bdd =
  bdd.identity

let nature_equal nature1 nature2 =
  match nature1, nature2 with
  | C b1, C b2 ->
      b1 = b2
  | D (x1, t1, f1), D (x2, t2, f2) ->
      x1 = x2 && equal t1 t2 && equal f1 f2
  | C _, D _
  | D _, C _ ->
      false

let combine x y = x * 65599 + y

let nature_hash nature =
  match nature with
  | C b ->
      if b then 1 else 0
  | D (x, t, f) ->
      combine x (combine (hash t) (hash f))

module N = Hashtbl.Make(struct
  type t = nature
  let equal = nature_equal
  let hash = nature_hash
end)
```

FIGURE 5 – Diagrammes binaires de décision (début)

```
let new_counter () : unit -> int =
  let c = ref 0 in
  fun () ->
    let i = !c in
    c := i + 1;
    i

let memo (f : nature -> 'a) : nature -> 'a =
  let table = N.create 1024 in
  fun x ->
    try
      N.find table x
    with Not_found ->
      let y = f x in
      N.add table x y;
      y

let construct : nature -> bdd =
  let next = new_counter() in
  memo (fun nature ->
    { identity = next(); nature = nature }
  )

let zero =
  construct (C false)

let one =
  construct (C true)

let var i =
  construct (D (i, one, zero))

let d i t f =
  if equal t f then t else construct (D (i, t, f))
```

FIGURE 6 – Diagrammes binaires de décision (suite)