Composition d'Informatique
Les Principes des Langages de Programmation (INF 321)
Promotion 2010

Sujet proposé par Eric Goubault

11 juillet 2011

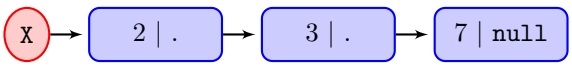
Les exercices et le problème qui suivent sont indépendants et peuvent être traités dans n'importe quel ordre. Cependant, la réponse à la question 2 de l'exercice 2 aide à répondre à la question 2.b de l'exercice 3. On attachera une grande importance à la clarté, à la précision et à la concision de la rédaction.

1 Exercice 1 (5 points)

On se donne la classe des listes d'entiers

```
class List {
 int hd;
 List tl;
 List(int h, List t) { hd = h; tl = t; }
}
```

Soit `N` un entier naturel strictement supérieur à 2, représenté par une *constante globale* Java. On souhaite écrire un programme Java `Prog` qui manipule des entiers naturels écrits en base `N`, sous forme de listes d'entiers. Un tel entier sera donc représenté par une variable `X` de type `List` qui décrit dans l'ordre, les digits en base `N`. Ainsi lorsque `X` référence la liste $(2, 3, 7)$, en base 16:



`X` représente l'entier $(((2) \times 16 + 3) \times 16 + 7 = 567$.

1. Comment déclarer la variable `N` dans la `class Prog`?
2. Quel nombre la liste référencée par `X` représente-t-elle lorsque `X` vaut `null`?
3. Ecrire une fonction `static boolean equal` qui prend deux listes d'entiers `X` et `Y` et renvoie `true` si `X` et `Y` représentent le même nombre, `false` sinon. On supposera que les listes non vides commencent toutes par un entier non nul.
4. Ecrire une fonction `static int base10(List X)` qui transforme `X` écrit en base `N` en sa valeur entière (donc en base 10). On supposera que les nombres manipulés sont toujours inférieurs à la taille maximale d'un `int`.

2 Exercice 2 (5 points)

On considère ici les tableaux `T` de n booléens: `boolean[] T;`

1. Ecrivez une instruction d'une ligne qui déclare une référence `T` à un tableau de booléens, alloue en mémoire un tableau de `n` booléens et affecte à `T` la référence au tableau créé.

2. Soit `n` un entier positif et `T` une référence à un tableau de booléens de taille *superieure ou egale a n*. Ecrire une fonction *recursive*

   ```
   static void imprimeEnumeration(int n, boolean[] T)
   ```

   qui imprime à l'écran toutes les modifications possibles du tableau référencé par `T` entre les cases `0` et `n-1` incluses, sans changer les autres cases. On supposera écrite une fonction:

   ```
   static void imprime(boolean[] S)
   ```

   qui imprime un tableau booléen référencé par `S`.

3. Quelle est la complexité en temps de la fonction `imprimeEnumeration`?

3 Problème (10 points+4 points questions optionnelles)

On se donne la classe d'arbres suivante, permettant de représenter des expressions booléennes

```
class BooleanExpr {
 int select;
 int Var;
 boolean Cste;
 BooleanExpr argg;
 BooleanExpr argd;
 BooleanExpr(int s, int v, boolean c, BooleanExpr ag, BooleanExpr ad) {
  select = s; Var = v; Cste = c; argg = ag; argd = ad;
 }
}
```

Soit `E` de type `BooleanExpr`. `E.select` donne le type d'expression booléenne que représente `E`:

- Si `E.select` vaut 0: `E` représente une variable booléenne x_i, $i = 0, \ldots, n-1$ (n fixé par ailleurs), et seul le champ `E.Var`, qui vaut i, est utile
- Si `E.select` vaut 1: `E` représente une constante booléenne, dont la valeur est donnée par le champ `E.Cste`
- Si `E.select` vaut 2: `E` est l'expression $\neg$`E.argg`, c'est-à-dire la négation de l'expression représentée par `E.argg`
- Si `E.select` vaut 3: `E` est l'expression `E.argg`$\wedge$`E.argd`, c'est-à-dire le *et logique* des expressions booléennes représentées par `E.argg` et `E.argd`.

1. Première partie (5 points)

   (a) Quelle est l'expression booléenne représentée par `x2` dans:

   ```
   x1 = new BooleanExpr(0,1,true,null,null);
   x2 = new BooleanExpr(2,0,false,x1,null);
   ```

   (b) Comment construire (en utilisant le constructeur de la classe `BooleanExpr` fourni dans l'énoncé) un objet de type `BooleanExpr` représentant l'expression booléenne

   $$\neg\left(x_0 \wedge (\neg x_1)\right)$$

   On pourra construire successivement $\neg x_1$ et $x_0 \wedge \neg x_1$ avant de construire l'expression $\neg\left(x_0 \wedge (\neg x_1)\right)$.

(c) Ecrire de nouveaux constructeurs:

```
BooleanExpr(int v)
BooleanExpr(boolean c)
BooleanExpr(BooleanExpr x, BooleanExpr y)
```

qui construisent respectivement les expressions: variable booléenne numéro `v`, constante booléenne égale à `c`, et le ***et logique*** des expressions `x` et `y`.

(d) Etant donné les égalités:

$$\begin{array}{rcll} x \vee y & = & \neg\left((\neg x) \wedge (\neg y)\right) & \text{ou logique} \\ x \Rightarrow y & = & (\neg x) \vee y & \text{implication logique} \end{array}$$

écrire des fonctions:

```
static BooleanExpr Non(BooleanExpr x)
static BooleanExpr Ou(BooleanExpr x, BooleanExpr y)
static BooleanExpr Implique(BooleanExpr x, BooleanExpr y)
```

retournant respectivement le ***non logique*** d'une expression booléenne `x`, le ***ou logique*** de deux expressions booléennes `x` et `y`, et l'***implication logique*** de `y` par `x` (`x`⇒`y`).

2. Deuxième partie (5 points). On s'autorise maintenant dans le type `BooleanExpr` à représenter explicitement l'opération $\vee$ (ou logique). Quand `E` est de type `BooleanExpr`, si `E.select` vaut 4, `E` est l'expression `E.argg`$\vee$`E.argd`, c'est-à-dire le ***ou logique*** des expressions booléennes représentées par `E.argg` et `E.argd`.

   Dans la suite, les valeurs des n variables booléennes $x_0, \ldots, x_{n-1}$ sont données par un tableau de n booléens `T`, comme à l'exercice 2.

   (a) Ecrire une fonction

   ```
   static boolean eval(BooleanExpr E, boolean[] T)
   ```

   qui évalue l'expression booléenne `E` avec les valeurs de vérité pour ses n variables données par le tableau `T` (on supposera que `T` a la bonne taille, n). On utilisera entre autres les règles:

$$\begin{array}{rcl} eval(true \vee E, T) & = & true \\ eval(false \vee E, T) & = & eval(E, T) \\ eval(false \wedge E, T) & = & false \\ eval(true \wedge E, T) & = & eval(E, T) \\ eval(\neg E, T) & = & \neg eval(E, T) \end{array}$$

   (b) En s'inspirant du code de la fonction d'énumeration des tableaux booléens de l'exercice 2, écrire une fonction:

   ```
   static boolean SAT(BooleanExpr E)
   ```

   qui renvoit `true` si `E` est vraie pour ***toutes*** les valeurs de ses n variables.

   (c) Etant donné une expression booléenne représentée par un arbre à k noeuds, quelle est la complexité dans le cas le pire, de son évaluation pour des valeurs de vérité données par un tableau à n entrées booléennes? En déduire dans le cas le pire, la complexité de la fonction `SAT` de la question précédente.

3. Troisième partie (optionnelle, 4 points):

   (a) On souhaite maintenant mettre l'expression booléenne `E`, codée par les arbres de la deuxième partie de ce problème, en forme NNF (***negational normal form***). Dans cette forme NNF, les formules ne contiennent des négations qu'aux ***feuilles*** des arbres les représentant. Ainsi,

$$\neg(x_0 \wedge (\neg x_1))$$

est, en forme NNF, équivalent à

$$\neg x_0 \vee x_1$$

L'idée de la transformation est donc de faire *descendre* les négations vers les feuilles de l'arbre en utilisant les règles:

$$\begin{array}{rcl} \neg(a \vee b) & \rightarrow & (\neg a) \wedge (\neg b) \\ \neg(a \wedge b) & \rightarrow & (\neg a) \vee (\neg b) \end{array}$$

et d'utiliser l'élimination de la double négation:

$$\neg\neg a = a$$

Ecrire une fonction *recursive*

```
static BooleanExpr NNF(BooleanExpr E}
```

qui prend une expression booléenne `E` et qui la met en forme NNF.

(b) La forme NNF ainsi obtenue n'est pas forcément simplifiée. On souhaite utiliser les règles:

$$\begin{array}{rcll} x_i \wedge x_i & \rightarrow & x_i & (rappel : x_i \text{ est une variable}) \\ x_i \vee x_i & \rightarrow & x_i & \\ true \wedge E & \rightarrow & E & \\ false \wedge E & \rightarrow & false & \\ true \vee E & \rightarrow & true & \\ false \vee E & \rightarrow & E & \end{array}$$

Ecrire une fonction utilisant ces règles et *uni Td [(EesF]TJ -72.4387(fo8791.1 0 Td88(ces)-388(r)28( )472(e*