INF3140 — Modélisation et spécification formelles de logiciels
Examen final (Automne 2008)

Durée: 13h30 – 16h30 (3 heures) **Documentation autorisée:** Toute documentation personnelle.

Nom: _____

Code permanent: | | | | | | | | | | | | |

Directives:

a. Répondez aux questions directement sur le questionnaire. Notez toutefois que pour certaines sous-questions, vous devez utiliser le verso de la feuille pour votre réponse.

b. N'écrivez rien dans la table à droite (réservée au correcteur).

1	/10
2	/15
3	/10
4	/10
5	/10
Bonus	/5
Total	/55

1. Tests unitaires pour un type immuable `bag` (10 pts)

Le module Spec 1 (p. 9) présente un type immuable (collection de valeurs), exprimé dans un style procédural (`TYPE` avec des `MESSAGE`s), pour des sacs (des multi-ensembles). Vous remarquerez qu'il s'agit d'une solution partielle (partielle parce que certaines opérations ont été omises) à l'exercice 1 du devoir 3.

Comme pour les ensembles, séquences et dictionnaires, on pourrait définir une mise en oeuvre de ces sacs en Java et les incorporer à la bibliothèque `TypesSpec`. Supposons donc qu'une telle mise en oeuvre ait été réalisée. Pour cet exercice, votre tâche est alors de définir un certain nombre (3–4) de cas de tests qui vont permettre de vérifier le bon fonctionnement des opérations suivantes : `add`, `"["`, `size` et `"IN"`.

Vous devez supposer que la façon de définir et utiliser ces sacs en Java est la même que pour les ensembles, séquences et dictionnaires. Entre autres, ceci signifie que des méthodes statiques sont disponibles (style procédural Spec) ainsi que des méthodes d'instance (style objet). Vous n'avez pas à développer des tests pour les deux formes ; vous choisissez plutôt la forme que vous préférez — mais soyez *consistant* (uniforme) dans votre style d'utilisation.

Vous pouvez aussi supposer que la méthode suivante de fabrication est disponible dans la classe `Constructeurs` :

```
/**
 * Constructeur statique pour un sac vide.
 * L'objet retourné est unique (toujours le même pour tous les appels).
 */
public static final <T> Bag<T> emptyBag() {
```

Finalement, vous pouvez simplement développer vos tests pour des sacs d'entiers (`Integer`) — c'est-à-dire, pas besoin de tester pour des instances génériques différentes : à cause de la façon dont les génériques sont mis en oeuvre en Java, si cela fonctionne pour un type, cela fonctionne pour n'importe quel type.

2. Type immuable pour des `bit_sets` (15 pts)

Le module Spec 2 (p. 10) définit un type immuable pour des `bit_set`s, lesquels permettent de représenter de façon compacte certains ensembles, à savoir, ceux dont les éléments peuvent être associés à une valeur entière comprise dans un intervalle `1..n`. Une représentation compacte est possible car on peut alors représenter un ensemble à l'aide d'une série de bits. Ainsi, pour un ensemble de taille $\leq$ 32, un unique mot mémoire d'une machine à 32 bits suffit pour représenter n'importe quel ensemble. Les opérations peuvent aussi être mises en oeuvre de façon efficace simplement à l'aide de manipulation de bits.

Par exemple, soit `bs` un `bit_set` de longueur `8`. Les bits qui sont à 1 (`true`) représentent un élément présent dans l'ensemble, alors que les bits qui sont à 0 (`false`) indiquent des éléments qui ne sont pas présents. La séquence de bits «`10001110`» représente donc l'ensemble `{1, 5, 6, 7}`.

[5] a) Une première façon de modéliser un `bit_set` est d'utiliser un attribut de type `sequence{boolean}`, tel que cela est fait dans le module Spec 2. Une autre façon possible consiste à utiliser comme modèle un *dictionnaire*, qui indique la valeur booléenne (définition) associée à chacun des numéros de bits (clé).

Si on utilise un tel dictionnaire comme modèle, quels devraient alors être le `MODEL` et l'`INVARIANT` appropriés, de façon à pouvoir spécifier les diverses opérations?

[10] b) On veut spécifier un certain nombre d'opérations additionnelles sur ces `bit_set`s, que vous devez spécifier en Spec *en utilisant le modèle suggéré à la sous-question précédente* (dictionnaire) :

i. `size` : Détermine le nombre d'éléments qui sont présents dans le `bit_set`, c'est-à-dire qui sont à `true` — ne pas confondre avec `length`, qui détermine le nombre maximum d'éléments possibles.

ii. `and` : Produit un nouveau `bit_set` qui inclut les bits qui sont à la fois dans le premier *et* dans le deuxième ensemble.

iii. `element_minimum` : Retourne l'élément minimum parmi les divers éléments qui sont présents.

3. Type mutable (style objet) pour des files de priorité (10 pts)

Le module Spec 3 (p. 11) présente un type mutable (classe d'objets), exprimé dans un style objet, pour des files de priorité qui incorporent tant l'aspect «niveau de priorité» que l'aspect «*FIFO*» (*First-In, First-Out*). Donc, lorsqu'on retire un élément de la file (`retirer`), on retire celui dont le niveau de priorité est le plus élevé *et qui a été ajouté en premier dans la file parmi les divers éléments de même priorité.*

Vous devez définir des méthodes ou messages (comme vous le désirez) pour les trois (3) opérations ci-bas — en indiquant une pré-condition appropriée, si nécessaire. De plus, dans chaque cas, vous devez indiquer à l'aide d'un commentaire (voir module Spec 3) la sorte de méthode/message.

i. `priorite_max` : Détermine le niveau maximum de priorité parmi les divers éléments présents.

ii. `taille` : Retourne le nombre total d'éléments présents dans la file (peu importe le niveau de priorité).

iii. `retirer_plusieurs` : Permet *d'obtenir et de retirer*, en une seule opération, tous les éléments de la file qui ont *le niveau maximal de priorité*, et ce de façon à préserver leur ordre d'insertion/arrivée.

(Pour cette question, répondez au verso de la feuille!)

4. Méthodes `generate` et `reduce` sur des séquences (10 pts)

Soit les objets Java définis dans le Code Java 1 (p. 12). Soit la séquence `seq` définie comme suit :

```
Sequence<String> seq = sequence( "ab", "1043", "bonjour", "", "54321" );
```

Pour chacune des expressions présentées ci-bas :

i) Donnez la valeur simplifiée de l'expression — en Spec ou en Java, comme vous préférez.

ii) Donnez une expression Spec équivalente *qui soit la plus simple possible.* Si nécessaire, vous pouvez introduire des concepts auxiliaires.

i. `seq.generate( tousChiffres, id ) )`

ii. `seq.generate( longueur ).generate( diviserPar2 ).reduce( plus )`

iii.
```
seq
.domain()
.generate( divisiblePar2, diviserPar2 )
.generate( new Expression<Integer,String>() {
      public String evaluer( Integer i ) {
           return seq.index(i);
      } } )
.generate( longueur )
.reduce( plus )
```

(Pour cette question, répondez au verso de la feuille!)

5. Mise en oeuvre procédurale vs. spécification logique de fonctions (10 pts)

Deux méthodes de tests pour deux (2) fonctions sur des ensembles ou séquences de chaînes de caractères sont présentées dans le code Java 2 (p. 13).

Pour chaque fonction, décrite informellement plus bas, mais dont les signatures sont déjà indiquées...

i. Donnez une mise en oeuvre Java, dans un style *procédural*, de la fonction. La fonction doit aussi, juste avant de retourner le résultat, faire un appel à une fonction de vérification de la post-condition.

ii. Donnez une mise en oeuvre Java d'une procédure `assertPostCondition_nomDeLaFonction` qui vérifie que le résultat produit par la fonction est correct. Cette procédure doit être écrite dans un style *logique* (avec quantificateurs, prédicats, expressions génératrices, opérations de réduction, etc.).

Indices : Vous pouvez, si cela est utile (?), réutiliser les objets définis au numéro précédent.

[5] a) Une fonction `tousDeLongueurN` qui, étant donné un ensemble de chaines de caractères, détermine si toutes les chaines sont de la longueur indiquée par le deuxième arguments.

```
boolean tousDeLongueurN( Set<String> st, int n )
```

```
void assertPostCondition_tousDeLongueurN_Set( final Set<String> st,
                                              final int n,
                                              final boolean resultat )
```

[5] b) Une fonction `elementsIndicesImpairs` qui, étant donné une séquence de chaînes de caractères, retourne la sous-séquence des éléments qui correspondent à des indices impairs.

```
Sequence<String> elementsIndicesImpairs( Sequence<String> seq )
```

```
void assertPostCondition_elementsIndicesImpairs( final Sequence<String> seq,
                                                 final Sequence<String> resultat )
```

[Bonus 5 pts] c) Une fonction `elementsDeLongueurN` qui, étant donné une séquence de chaînes de caractères spécifiée par le premier argument, retourne l'ensemble des chaines qui sont de la longueur spécifiée par le deuxième argument. (Pour cette question, répondez au verso de la feuille!)

Modules Spec et code Java

```
TYPE bag{ t: type SUCH THAT Subtype(t, total_order{t}) }

  INHERIT total_order{bag{t}}
  IMPORT Subtype FROM type

  MODEL( elems: map{t, nat} )

  INVARIANT
    ALL( b: bag{t} :: default(b.elems) = 0 )

  MESSAGE empty_bag
    REPLY( b: bag{t} )
      WHERE b.elems = create(0)

  MESSAGE add ( x: t, b1: bag{t} )
    REPLY( b2: bag{t} )
      WHERE
        b2.elems = bind( x, b1.elems[x]+1, b1.elems )

  MESSAGE remove( x: t, b1: bag{t} )
    REPLY( b2: bag{t} )
     WHERE
       b2.elems = bind( x, 0 MAX b1.elems[x]-1, b1.elems )

  MESSAGE "["( b: bag{t}, x: t )
    REPLY( n: nat )
      WHERE n = b.elems[x]

  MESSAGE size( b: bag{t} )
    REPLY( n: nat )
       WHERE
         n = SUM( x: t SUCH THAT x IN domain(b.elems) :: b.elems[x] )

  MESSAGE "IN"( x: t, b: bag{t} )
    REPLY( r: boolean )
     WHERE
       r <=> b.elems[x] > 0
END
```

Module Spec 1: Un type immuable définissant des sacs (multi-ensembles).

```
TYPE bit_set

  MODEL( bits: sequence{boolean}, maxBits: nat )
  INVARIANT
    ALL( bs: bit_set :: bs.maxBits > 0 & length(bs.bits) = bs.maxBits )

  MESSAGE empty_bit_set( n: nat SUCH THAT n > 0 )
    REPLY( bs: bit_set )
      WHERE
        bs.bits = [i: nat SUCH THAT 1 <= i <= n :: false],
        bs.maxBits = n

  MESSAGE get_bit( bs: bit_set, i: nat )
    WHEN 1 <= i <= size(bs)
      REPLY( b: boolean )
        WHERE b = bs.bits[i]
    OTHERWISE
      REPLY EXCEPTION index_invalide

  MESSAGE set_bit( bs: bit_set, i: nat, bit: boolean )
    WHEN 1 <= i <= size(bs)
      REPLY( bs2: bit_set )
        WHERE bs2.bits = bs.bits[1..i-1] || [bit] || bs.bits[i+1..size(bs)]
    OTHERWISE
      REPLY EXCEPTION index_invalide

  MESSAGE length( bs: bit_set )
    REPLY( n: nat )
      WHERE n = bs.maxBits

  -- Concept auxiliaire.
  CONCEPT compatibles( bs1 bs2: bit_set )
    VALUE( b: boolean )
      WHERE b <=> length(bs1.bits) = length(bs2.bits)

END
```

Module Spec 2: Un type immuable pour des `bit_sets`.

```
CLASS file_priorite{ t: type SUCH THAT Subtype(t, priorite{t}) }
  IMPORT Subtype FROM type

  MODEL( elems: map{nat, sequence{t}} )
  INVARIANT default(elems) = []

  CREATOR file_priorite  -- Constructeur/createur
    INITIALLY
      elems = create([])

  METHOD ajouter( x: t )  -- Mutateur
    TRANSITION
      elems[priorite(x)] = *elems[priorite(x)] || [x]

  METHOD retirer()  -- Mutateur
    WHEN ~self->est_vide()
      CHOOSE( prio: nat SUCH THAT prio = self->priorite_max() )
      REPLY( x: t )
        WHERE
          priorite(x) = prio,
          x = *elems[prio][1]
      TRANSITION
        elems[prio] = *elems[prio][2..length(*elems[prio])]
    OTHERWISE
      REPLY EXCEPTION file_vide

  METHOD est_vide()  -- Observateur
    REPLY( b: boolean )
      WHERE b <=> domain(elems) = {}
END
```

Module Spec 3: Un type mutable pour des objets `file_priorite` qui incorporent à la fois l'aspect «niveau de priorité» et l'aspect «*FIFO*» (*First-In, First-Out*).

```
Expression<Integer,Integer> diviserPar2 = new Expression<Integer,Integer>() {
   public Integer evaluer( Integer x ) {
       return x / 2;
   }
};

Predicat<Integer> divisiblePar2 = new Predicat<Integer>() {
   public boolean evaluer( Integer x ) {
       return( x % 2 == 0 );
   }
};

Expression<String,String> id = new Expression<String,String>() {
   public String evaluer( String s ) {
       return s;
   }
};

Expression<String,Integer> longueur = new Expression<String,Integer>() {
   public Integer evaluer( String s ) {
       return s.length();
   }
};

OpBinaire<Integer> plus = new OpBinaire<Integer>() {
       public Integer evaluer( Integer v1, Integer v2 ) {
          return v1 + v2;
       }
   };

Predicat<String> tousChiffres = new Predicat<String>() {
   public boolean evaluer( String s ) {
       for( int i = 0; i < s.length(); i++ ) {
          if ( s.charAt(i) < '0' || s.charAt(i) > '9' ) {
              return false;
          }
       }
       return true;
   }
};
```

Code Java 1: Divers objets de type `Expression`, `Predicat` et `OpBinaire`, présentés en ordre alphabétique du nom de la variable.

```
@Test public void exemples_tousDeLongueurN() {
  Set<String> empty = Constructeurs.emptySet();

  assertTrue ( tousDeLongueurN( empty, 34 ) );
  assertTrue ( tousDeLongueurN( empty.add( "ab" ).add( "de" ), 2 ) );
  assertFalse( tousDeLongueurN( empty.add( "" ).add( "de" ), 2 ) );
}




@Test public void exemples_elementsIndicesImpairs() {
  assertEquals( sequence(),
                elementsIndicesImpairs( sequence() ) );

  assertEquals( sequence( "ab" ),
                elementsIndicesImpairs( sequence( "ab" ) ) );

  assertEquals( sequence( "ab" ),
                elementsIndicesImpairs( sequence( "ab", "1043" ) ) );

  assertEquals( sequence( "ab", "bonjour", "54321" ),
                elementsIndicesImpairs( sequence( "ab", "1043", "bonjour", "", "54321" ) ) );
}
```

Code Java 2: Méthodes de test pour des fonctions sur des séquences de chaînes de caractères.