Introduction à l'informatique II (CSI 1501)
Examen final

Professeur: Marcel Turcotte

Avril 2003, durée: 3 heures

Identification

Nom de l'étudiant(e) : _____

Numéro d'étudiant : _____ place désignée : _____

Consignes

1. C'est un examen à livres fermés.
2. Aucune calculatrice ou toute autre forme d'aide n'est permise.
3. Présentez le détail de vos solutions.
4. Écrivez lisiblement.
5. Ne retirez pas l'agrafe.
6. Répondez sur ce questionnaire, utilisez le verso des pages si nécessaire, mais vous ne pouvez remettre aucune page additionnelle.

Barème

Question	Maximum	Résultat
1	5	
2	4	
3	10	
4	6	
5	4	
6	10	
7	14	
8	15	
9	17	
10	15	
Total	**100**	

Question 1 *(5 points)*

Pour chacune des lignes de cette démonstration, indiquez, dans l'espace réservé à cet effet, l'axiome ou le théorème qui a été utilisé afin de transformer l'expression précédente en celle-ci. Consultez l'Appendice A pour trouver la liste complète des axiomes et théorèmes de l'algèbre de Boole vus en classe.

$$(\overline{x} \cdot y) + (x \cdot \overline{y}) + (x \cdot y)$$

$$(\overline{x} \cdot y) + (x \cdot \overline{y}) + (x \cdot y) + (x \cdot y) \quad (\_\_\_\_\_)$$

$$(\overline{x} \cdot y) + (x \cdot y) + (x \cdot \overline{y}) + (x \cdot y) \quad (\_\_\_\_\_)$$

$$y \cdot (\overline{x} + x) + x \cdot (\overline{y} + y) \quad (\_\_\_\_\_)$$

$$y \cdot 1 + x \cdot 1 \quad (\_\_\_\_\_)$$

$$y + x \quad (\_\_\_\_\_)$$

$$\square \text{ Q.E.D.}$$

Question 2 *(4 points)*

Donnez le Produit Canonique des Sommes (*Canonical Product of Sums* – CPOS) pour la table de vérité suivante. Ne simplifiez pas votre réponse.

x	y	z	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$F =$

Question 3 *(10 points)*

Traduisez les énoncés Java suivants en langage d'assemblage pour l'ordinateur TC-1101, présenté en classe. Consultez l'Appendice B afin de connaître la liste complète des mnémoniques de ce langage d'assemblage.

```
int a = 1;
int b = 1;
int n = 5;
for (int i=3; i <= n; i++) {
    int c = a + b;
    a = b;
    b = c;
}
System.out.println(b);
```

Question 4 *(6 points)*

Pour cette question, **six (6) bits** sont utilisés afin de représenter des entiers binaires signés à l'aide de la représentation en complément à deux.

a) Représentez l'entier signé $(-26)_{10}$ en complément à deux.

Réponse : [_____]$_2$

b) À l'aide de l'arithmétique binaire en complément à deux, calculez la somme suivante.

$$\begin{array}{rcccccc} & 1 & 1 & 0 & 1 & 1 & 0 \\ + & 0 & 1 & 1 & 0 & 1 & 1 \\ \hline \end{array}$$

c) Donnez la représentation binaire **ainsi que** la représentation décimale, de la plus petite **et** de la plus grande valeur encodée à l'aide de six bits et de la représentation en complément à deux.

- plus petite valeur [_____]$_2$, (_____)$_{10}$
- plus grande valeur [_____]$_2$, (_____)$_{10}$

Note : la plus petite valeur, dans le cas d'un nombre signé, est négative.

Question 5 *(4 points)*

a) Convertissez le nombre binaire sans signe suivant en octal (notez qu'il est possible de convertir un nombre binaire en octal de façon simple, nous avons vu en classe une telle méthode).

$$(1010110.11)_2 = (\_\_\_\_\_)_8$$

b) Convertissez le nombre décimal sans signe suivant en binaire.

$$(45.3125)_{10} = (\_\_\_\_\_)_2$$

Question 6 *(10 points)*

Le problème de la patate chaude implique N personnes, numérotées de 0 à $(N-1)$, et une patate! Les N personnes forment un cercle et chaque personne passe la patate à son voisin immédiat. Suite à M passes, la personne ayant la patate en mains est éliminée. La dernière personne restante gagne. Le problème consiste donc à déterminer le numéro de la personne gagnante lorsqu'il y a N participants et que M passes sont faites pour chaque ronde. Au début du jeu, la personne numéro 0 reçoit la patate. Pour toutes les autres rondes, la personne suivant celle éliminée a la patate au début du tour. Ainsi, pour $N = 3$ et $M = 4$, le gagnant porte le numéro 0, alors que pour $N = 4$ et $M = 3$, le gagnant sera le 1.

Avec quelques efforts, on peut résoudre ce problème mathématiquement. Ici, nous avons choisi d'écrire un programme simulant ce jeu et permettant de déterminer le numéro du vainqueur. Vous devez compléter l'implémentation partielle ci-bas. En particulier, il vous faudra choisir entre une file et une pile pour implémenter votre solution. Vous pouvez présumer l'existence d'une implémentation pour chacune des deux interfaces suivantes, par exemple, **LinkedQueue** et **LinkedStack**.

```
public interface Queue {
    public abstract void enqueue(Object obj);
    public abstract Object dequeue();
    public abstract boolean isEmpty();
}

public interface Stack {
    public abstract Object push(Object obj);
    public abstract Object pop();
    public abstract boolean isEmpty();
}

public static void solve(int n, int m) {

    ____________ x = new ________________________;

    for (int i=0; i<n; i++) {
        Object person = new Integer(i);

        x . ____________ (person);
    }

    Object last = null;

    while (! x.isEmpty()) {

        for (int i=0; i<m; i++) {

            Object person = ___________________;

            ___________________;
        }

        last = ___________________;
    }
    System.out.println("last = " + last);
}
```

Question 7 *(14 points)*

Le type abstrait de données Deque — qui se prononce "deck" — a les caractéristiques d'une file et d'une pile. En particulier, une structure Deque ("Double-Ended QUEue") nous assure que,

– les ajouts à l'avant ou à l'arrière de la file sont efficaces ;
– les délétions à l'avant ou à l'arrière de la file sont efficaces.

Une structure de données Deque est généralement implémentée à l'aide d'un tableau circulaire. Sur la page suivante, vous trouverez une implémentation partielle de la classe Deque qui 1) utilise un tableau circulaire afin de sauvegarder les éléments et 2) utilise une variable d'instance, **size**, afin de distinguer une structure vide, d'une structure pleine. Voici la description des quatre méthodes principales de cette classe.

Object push(Object item) ; ajoute un item à l'**arrière** de la file et retourne la valeur de cet item.

Object pop() ; retire l'élément se trouvant à l'**arrière** de la file et retourne la valeur de cet élément.

Object unshift(Object item) ; ajoute un item à l'**avant** la file et retourne la valeur de cet élément.

Object shift() ; retire l'élément se trouvant à l'**avant** de la file et retourne sa valeur.

Pour l'implémentation partielle de la classe Deque se trouvant à la page suivante, complétez l'implémentation des méthodes **Object unshift(Object item)**, et **Object shift()**.

Question bonus (2 points)

La méthode **Object push(Object obj)** utilise une expression arithmétique modulo telle que celle-ci,

```
rear = (rear + 1) % MAX_DEQUE_SIZE;
```

afin d'implémenter le retour au début du tableau de l'index arrière lorsque la valeur de celui-ci excède celle du plus haut index. Cet énoncé pourrait être remplacé par les énoncés suivants,

```
rear = rear + 1;
if (rear == MAX_DEQUE_SIZE) {
    rear = 0;
}
```

La méthode **Object pop()** utilise un énoncé **if** comme suit afin d'implémenter le retour vers la fin du tableau lorsque la valeur de l'index arrière devient négative,

```
rear = rear - 1;
if (rear == -1) {
    rear = MAX_DEQUE_SIZE - 1;
}
```

Quelle expression arithmétique modulo pourrait remplacer les énoncés ci-haut, afin d'implémenter le retour vers la fin du tableau lorsque la valeur de l'index devient négative.

Réponse:

```
import java.util.NoSuchElementException;

public class Deque {

    private static final int MAX_DEQUE_SIZE = 8;
    private Object[] elems;
    private int front, rear, size;

    public Deque() {
        elems = new Object[MAX_DEQUE_SIZE];
        rear = front = size = 0;
    }

    public int size() { return size; }

    public boolean isEmpty() { return size == 0; }

    public boolean isFull() { return size == MAX_DEQUE_SIZE; }

    public Object push(Object obj) {
        if (size == MAX_DEQUE_SIZE)
            throw new IllegalStateException("deque is full");

        rear = (rear + 1) % MAX_DEQUE_SIZE;
        elems[rear] = obj;
        size++;
        return obj;
    }

    public Object pop() {
        if (size == 0)
            throw new NoSuchElementException("deque is empty");

        Object obj = elems[rear];
        elems[rear] = null;

        rear = rear - 1;

        if (rear == -1)
            rear = MAX_DEQUE_SIZE - 1;

        size--;
        return obj;
    }

    public Object shift() {

    } // end of Object shift()

    public Object unshift(Object obj) {

    } // end of Object unshift(Object obj)
} // end of class Deque
```

Question 8 *(15 points)*

Il y a un algorithme simple permettant de calculer le complément à deux d'un nombre binaire directement. Cet algorithme a deux étapes.

- À partir du bit le moins significatif, copier tous les bits jusqu'à ce que le premier 1 ait été copié ;
- Après que le premier 1 ait été copié, remplacer chacun des bits restants par leur complément, autrement dit remplacer chaque 0 par un 1 et chaque 1 par un 0.

Exemples :

- $(0101100)_2 = [1010100]_2$
- $(010111)_2 = [101001]_2$

où $[N]_2$ représente le complément à deux d'un nombre.

Vous devez implémenter cet algorithme à l'aide des itérateurs de Java et de l'implémentation partielle d'une liste de bits décrite ci-bas et présentée aux pages suivantes.

- Contrairement à l'ensemble des implémentations de liste présentées en classe, le type des éléments de la liste est **int**.
- Chaque élément de la liste est soit 0 ou 1.
- Les bits sont sauvegardés dans l'ordre inverse, c'est-à-dire que le bit le moins significatif est celui se trouvant dans le premier noeud de la liste. Ainsi, la suite de bits 10011 sera représentée comme suit,

  ```
  -> 1 -> 1 -> 0 -> 0 -> 1
  ```
- La liste vide équivaut à la liste ne contenant qu'un seul 0.
- La méthode d'instance **iterator()** retourne un objet implémentant les méthodes de l'interface **Iterator**. Notez que la méthode **next()** retourne une valeur de type **int**, ce qui est consistant avec le fait que les éléments de la liste sont des entiers.

  ```
  public interface Iterator {
      public boolean hasNext();
      public int next();
      public void add(int value);
  }
  ```

Vous devez compléter l'implémentation de la méthode d'instance **BitList twosComplement()**. L'implémentation doit être itérative (c'est-à-dire qu'il faut utiliser des itérateurs afin de traverser la liste). La méthode retourne une nouvelle instance de **BitList** et ne doit pas modifier *cet* instance. Étant donné une instance **b** représentant la liste de bits, 0101100,

```
-> 0 -> 0 -> 1 -> 1 -> 0 -> 1 -> 0
```

l'appel de méthode **b.twosComplement()** doit retourner une nouvelle instance de **BitList** contenant les bits suivants,

```
-> 0 -> 0 -> 1 -> 0 -> 1 -> 0 -> 1
```

et **b** demeure inchangé.

```
import java.util.NoSuchElementException;
import java.util.ConcurrentModificationException;

public class BitList {

    public static final int ZERO = 0;
    public static final int ONE = 1;

    private static class Node {
        private int value;
        private Node next;

        private Node(int value, Node next) {
            this.value = value;
            this.next  = next;
        }
    }

    private Node first = null;
    private byte  modCount = 0;

    public BitList() {  }

    public BitList(String s) {

        if (s == null)
            return;

        for (int i=0; i < s.length() ; i++) {
            char c = s.charAt(i);
            addFirst(c - '0'); // renverse l'ordre
        }
    }

    public void addFirst(int value) {
        if ((value != 0) && (value != 1))
            throw new IllegalArgumentException();

        first = new Node(value, first);
        modCount++;
    }

    public Iterator iterator() {
        return new BitListIterator();
    }
```

```
// une classe imbriquée non-statique (inner class)

private class BitListIterator implements Iterator {

    private Node current = null;
    private int expectedModCount = modCount;

    public int next() {
        checkValid();

        if (current == null) {
            current = first;
        } else {
            current = current.next; // déplace le curseur
        }
        if (current == null)
            throw new NoSuchElementException();

        return current.value;
    }

    public void add(int newElement) {
        Node newNode;
        if (current == null) {
            first = new Node(newElement, first);
            current = first;
        } else {
            current.next = new Node(newElement, current.next);
            current = current.next;
        }

        modCount++;
        expectedModCount++;
    }

    public boolean hasNext() {
        return ((current == null) && (first != null))  ||
            ((current != null) && (current.next !=null));
    }

    private void checkValid() {
        if (expectedModCount != modCount)
            throw new ConcurrentModificationException();
    }
}
```

```
    public BitList twosComplement() {

    } // end of twosComplement()
} // end of BitList
```

Question 9 *(17 points)*

Cette question porte sur un langage simple afin d'évaluer des expressions arithmétiques. Ce langage est en fait un sous-ensemble du langage PostScript, que certaines imprimantes utilisent. La principale structure de données d'un interprète PostScript est la pile des opérandes. Pour l'interprète présenté à la page qui suit, vous devez implémenter les opérations **sub**, **exch** et **pstack**. Voici les 6 opérations du langage.

add : retire deux éléments éléments du dessus de la pile des opérandes, en fait leur somme, et remet le résultat sur le dessus de la pile.

sub : retire les deux éléments du dessus de la pile des opérandes, effectue la soustraction, et remet le résultat sur le dessus de la pile. Ainsi, l'expression $(3 - 1)$ serait représentée par l'expression PostScript suivante "`3 1 sub`".

eq : retire les deux éléments du dessus de la pile des opérandes, les compare, et remet sur le dessus de la pile la valeur booléene appropriée.

exch : échange l'ordre des deux éléments se trouvant au dessus de la pile.

dup : insère une copie de l'élément se trouvant au dessus de la pile.

pstack : imprime le contenu de la pile. Il est important que le contenu de la pile demeure inchangé suite à un appel à cette méthode. Utilisez le format de l'exemple ci-bas.

L'exécution du programme PostScript suivant,

```
> java Run "3 pstack dup pstack add pstack"
```

produira la sortie suivante,

```
-top-
INTEGER: 3
-bottom-
-top-
INTEGER: 3
INTEGER: 3
-bottom-
-top-
INTEGER: 6
-bottom-
```

La classe **Calculator**, présentée dans les pages qui suivent, est un interprète pour ce langage. L'implémentation nécessite trois classes supplémentaires : **Stack**, **Token** et **Reader**. La classe **Stack** implémente les trois méthodes suivantes :

boolean isEmpty() ; retourne vrai si cette pile ne contient aucun élément.

Object pop() ; retire l'élément du dessus de la pile et retourne sa valeur.

Object push(Object element) ; ajoute un élément sur le dessus de la pile et retourne sa valeur.

```
import Stack;

public class Calculator {

    private Stack operands;

    public Calculator() {
        operands = new Stack();
    }

    public void execute(String program) {

        Reader r = new Reader(program);

        while (r.hasMoreTokens()) {

            Token t = r.nextToken();

            if (! t.isSymbol()) {

                operands.push(t);

            } else if (t.sValue().equals("add")) {

                // implémentation de l'opération ``add''

            } else ... {

                // voir à la page suivante

            }
        }
    }
}
```

L'argument de la méthode **execute** est une chaîne de caractères représentant un programme PostScript valide, par exemple, "`1 pstack dup pstack`". La classe **Reader** traite cette chaîne et retourne un jeton à la fois. Ainsi, le premier appel à la méthode **nextToken()** retournera un jeton représentant le 1, puis le second appel retournera un jeton représentant le symbole `pstack`, et ainsi de suite. Les jetons qui ne sont pas des symboles sont mis sur la pile des opérandes alors que les symboles sont évalués immédiatement. Voici une brève description des classes **Token** et **Reader**.

Token : Un jeton est un objet immuable qui représente soit une valeur booléene, un entier ou un symbol.

Reader : Découpe une chaîne de caractères en jetons.

Sur la page suivante, complétez l'implémentation des opérations **sub**, **exch** et **pstack**.

```
    public void execute(String program) {
        Reader r = new Reader(program);
        while (r.hasMoreTokens()) {
            Token t = r.nextToken();
            if (! t.isSymbol()) {
                operands.push(t);
            } else if (t.sValue().equals("add")) {
                Token op1 = (Token) operands.pop();
                Token op2 = (Token) operands.pop();
                Token res = new Token(op1.iValue() + op2.iValue());
                operands.push(res);
            } else if (t.sValue().equals("sub")) { // complétez









            } else if (t.sValue().equals("eq")) {
                Token op1 = (Token) operands.pop();
                Token op2 = (Token) operands.pop();
                Token res = new Token(op1.iValue() == op2.iValue());
                operands.push(res);
            } else if (t.sValue().equals("exch")) { // complétez









            } else if (t.sValue().equals("dup")) {
                Token op = (Token) operands.pop();
                operands.push(op);
                operands.push(op);
            } else if (t.sValue().equals("pstack")) { // complétez









            } else {
                System.out.println("not a valid symbol");
            }
        }
    }
}
```

```
class Token {
    private static final int BOOLEAN = 0;
    private static final int INTEGER = 1;
    private static final int SYMBOL = 2;

    private boolean bValue;
    private int iValue;
    private String sValue;
    private int type;

    Token(boolean bValue) {
        this.bValue = bValue;
        type = BOOLEAN;
    }
    Token(int iValue) {
        this.iValue = iValue;
        type = INTEGER;
    }
    Token(String sValue) {
        this.sValue = sValue;
        type = SYMBOL;
    }
    public boolean bValue() {
        if (type != BOOLEAN)
            throw new IllegalStateException("not a boolean");
        return bValue;
    }
    public int iValue() {
        if (type != INTEGER)
            throw new IllegalStateException("not an integer");
        return iValue;
    }
    public String sValue() {
        if (type != SYMBOL)
            throw new IllegalStateException("not a symbol");
        return sValue;
    }
    public boolean isBoolean() { return type == BOOLEAN;  }
    public boolean isInteger() { return type == INTEGER;  }
    public boolean isSymbol()  { return type == SYMBOL;   }
    public String toString() {
        switch (type) {
        case BOOLEAN: return "BOOLEAN: " + bValue;
        case INTEGER: return "INTEGER: " + iValue;
        case SYMBOL:  return "SYMBOL: "  + sValue;
        default:      return "INVALID";
        }
    }
}
```

```
class Reader {
    private StringTokenizer st;

    Reader(String s) {
        st = new StringTokenizer(s);
    }
    public boolean hasMoreTokens() {
        return st.hasMoreTokens();
    }
    public Token nextToken() {
        String t = st.nextToken();

        if ("true".equals(t))
            return new Token(true);

        if ("false".equals(t))
            return new Token(false);

        try {
            return new Token(Integer.parseInt(t));
        } catch (NumberFormatException e) {
            return new Token(t);
        }
    }
}

public class Run {
    public static void main(String[] args) {
        Calculator c = new Calculator();
        for (int i=0; i<args.length; i++) {
            c.execute(args[i]);
        }
    }
}
```

Question 10 *(15 points)*

Écrivez une méthode **récursive** qui retire et retourne le $i$ème élément d'une liste simplement chaînée. Les positions de la liste sont numérotées de 0 à $(n-1)$. Ainsi, pour une instance **l** qui contient les éléments a, b, c et d, l'appel de méthode **l.remove(2)** retourne l'élément c et transforme la liste de sorte que l'élément c ait été retiré, et que le contenu de la liste soit maintenant a, b et d.

La méthode **public Object remove(int index)** doit être implémentée selon l'approche vue en classe, où les méthodes récursives ont une partie publique et une partie privée. La partie publique est responsable des mises à jour de l'entête de la liste et initie le premier appel récursif. Vous devez compléter la définition de la méthode privée **Object remove(Node p, int index)** (se trouvant à la page suivante) pour l'implémentation d'une liste simplement chaînée ci-bas.

La méthode doit lancer une exception de type **IndexOutOfBoundsException** si nécessaire. Cette méthode ne peut faire appel à aucune autre méthode qu'elle même pour effecter son travail — vous ne pouvez assumer l'existence d'aucune autre méthode.

```
public class SinglyLinkedList {

    private static class Node {
        Object value;
        Node next;

        Node(Object value, Node next) {
            this.value = value;
            this.next = next;
        }
    }

    private Node head = null;

    public void addFirst(Object obj) {
        if (obj == null)
           return false;
        head = new Node(obj, head);
        return true;
    }

    // voir page suivante
```

```
    public Object remove(int index) {

        if (head == null || index < 0)
            throw new IndexOutOfBoundsException();

        Object result;

        if (index == 0) {
            result = head.value;
            head = head.next;
        } else {
            result = remove(head, index-1);
        }

        return result;
    }

    private Object remove(Node p, int index) {

    } // end of private Object remove(Node p, int index)
} // end of SinglyLinkedList
```

Appendice A : algèbre de Boole

Les axiomes et théorèmes de l'algèbre booléenne. Étant donné un ensemble $\mathcal{B}$, contenant au moins deux éléments, 0 et 1, ainsi que deux opérations binaires suivantes, $+$ et $\cdot$ (addition et produit), et un opérateur unaire, $\overline{a}$ (négation). L'ensemble $\mathcal{B}$ est une algèbre de Boole si les opérations sont fermées sur $\mathcal{B}$,

$$a + b \in \mathcal{B}$$

$$a \cdot b \in \mathcal{B}$$

$$\overline{a} \in \mathcal{B}$$

et vérifient les axiomes suivants pour tout $a, b, c \in \mathcal{B}$,

Axiome 1 *fermeture :*

$$a + b \in \mathcal{B}$$

$$a \cdot b \in \mathcal{B}$$

$$\overline{a} \in \mathcal{B}$$

Axiome 2 *commutativité :*

$$a + b = b + a$$

$$a \cdot b = b \cdot a$$

Axiome 3 *associativité :*

$$(a + b) + c = a + (b + c)$$

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Axiome 4 *distributivité :*

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

$$a + (b \cdot c) = (a + b) \cdot (a + c)$$

Axiome 5 *éléments neutres, il existe* $0, 1 \in \mathcal{B}$ *tels que :*

$$a + 0 = a$$

$$a \cdot 1 = a$$

Axiome 6 *complémentation, pour tout* $a \in \mathcal{B}$*, il existe un* $\overline{a}$ *tel que :*

$$a + \overline{a} = 1$$

$$a \cdot \overline{a} = 0$$

Les théorèmes suivants peuvent être démontrés à partir des axiomes ci-haut.

idempotence	$a \cdot a = a$	$a + a = a$
élément absorbant	$a \cdot 0 = 0$	$a + 1 = 1$
absorption	$a \cdot (a + b) = a$	$a + a \cdot b = a$
de Morgan	$\overline{a \cdot b} = \overline{a} + \overline{b}$	$\overline{a + b} = \overline{a} \cdot \overline{b}$
involution	$\overline{(\overline{a})} = a$	

Appendice B : langage d'assemblage TC-1101

Mnémonique	opCode	Description
LDA	91	load x
STA	39	store x
CLA	08	clear (a=0, z=vrai, n=faux)
INC	10	incémente accumulateur (modifie z et n)
ADD	99	ajoute MDR à l'accumulaeur (modifie z et n)
SUB	61	retranche MDR de l'accumulateur (modifie z et n)
JMP	15	branchement inconditionnel vers x
JZ	17	branchement sur x si z==vrai
JN	19	branchement sur x si n==vrai
DSP	01	affiche la valeur se trouvant à l'adresse x
HLT	64	fin

(page blanche)