Introduction to Computer Science II (CSI 1101)
Final Examination

Instructor: Marcel Turcotte

April 2003, duration: 3 hours

Identification

Student name (firstname, surname): ______________________________

Student number: ____________________ designated seat: ____________________

Instructions

1. This is a closed book examination.
2. No calculators or other aids are permitted.
3. Write comments and assumptions to get partial marks.
4. Beware, poor hand writing can affect grades.
5. Do not remove the staple holding the examination pages together.
6. Write your answers in the space provided. Use the backs of pages if necessary. You may **not** hand in additional pages.

Marking scheme

Question	Maximum	Actual
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 marks)*

Next to each line of the following demonstration, write down the axiom or theorem that was used to transform the preceding expression into this one. Consult the Appendix A for the complete list of all the axioms and theorems seen in class.

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

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

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

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

$$y \cdot 1 + x \cdot 1 \quad (\underline{\qquad\qquad\qquad\qquad})$$

$$y + x \quad (\underline{\qquad\qquad\qquad\qquad})$$

$\square$ Q.E.D.

Question 2 *(4 marks)*

Give the Canonical Product of Sums (CPOS) for the following truth table. Do not simplify your answer.

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 marks)*

Convert the following Java program into the assembly language for the toy computer, TC-1101, presented in class. Consult the Appendix B for the list of all the mnemonics of this assembly language.

```
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 marks)*

For this question, **six (6) bits** are used to represent **signed** binary integers using **two's complement**.

a) Represent the signed decimal integer $(-26)_{10}$ in two's complement.

Answer: $[\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_]_2$

b) Use two's complement arithmetic to calculate the following sum.

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

c) Give the binary representation **and** the decimal value, of the smallest **and** the largest possible value encoded using **six bits** in **two's complement**.

- smallest value $[\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_]_2$, $(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)_{10}$
- largest value $[\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_]_2$, $(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)_{10}$

Hint: in the case of a signed number, the smallest value is negative.

Question 5 *(4 marks)*

a) Convert the following unsigned binary number to octal (notice that it is possible to convert directly from binary to octal, we have seen a method in class that does just that).

$$(1010110.11)_2 = (\underline{\hspace{5cm}})_8$$

b) Convert the following unsigned decimal number to binary.

$$(45.3125)_{10} = (\underline{\hspace{5cm}})_2$$

Question 6 *(10 marks)*

The Josephus problem (named after Flavious Josephus — an historian from the first century) — consists of N people, numbered 0 to $(N - 1)$, and a potato. The N persons are sitting in a circle. During a round, each person passes the potato to his/her immediate neighbour. Each round consists of M passes. After M passes, the person holding the potato is eliminated. The last remaining person wins. The problem is therefore to determine the number of the last person remaining, assuming N participants and M passes are made before a participant is eliminated. The game starts with the person number 0 having the potato. For subsequent rounds, the person next to the person that was eliminated starts with the potato. For $N = 3$ and $M = 4$ the number of the last person remaining would be 0, and for $N = 4$ and $M = 3$ the number would be 1.

With some efforts, the problem can be solve mathematically. Here, we choose to write a program to simulate the game and determine the number of the last person remaining. Complete the partial implementation below. In particular, this will require you to select between a queue or a stack data structure to implement your solution — you can assume the existence of an implementation for each of the following interfaces, for example, **LinkedQueue** and **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 marks)*

The abstract data type Deque — pronounced “deck” — combines features of both a queue and a stack. In particular, a Deque (“Double-Ended QUEue”) allows for,

- efficient insertions at the front or rear of the queue;
- efficient deletions at the front or the rear of the queue

Generally, a Deque is implemented with a circular array. On the next page, you will find a partial implementation of the class Deque that 1) uses a circular array to store the elements and 2) uses an instance variable, **size**, to distinguish between a Deque that is full or empty. Here is the description of the four main methods of this class.

Object push(Object item); adds an item at the **rear** of this queue and returns the item argument.

Object pop(); removes the object at the **rear** of this queue and returns that object as the value of this method.

Object unshift(Object item); adds an item at the **front** of this queue and returns the item argument.

Object shift(); removes the object at the front of this **queue** and returns that object as the value of this method.

In the partial implementation of the class Deque on the next page, complete the implementation of the methods **Object unshift(Object item)**, and **Object shift()**.

Bonus question (2 marks)

The method **Object push(Object obj)** uses a modulo arithmetic expression such as this one,

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

to implement the wraparound of the rear index on the right hand side of the array. This could also have been implemented as follows,

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

The method **Object pop()** uses an **if** statement to implement the wraparound when the value of the index **rear** becomes negative,

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

What modulo arithmetic expression can be used to replace the above **if** statement and to implement the wraparound on the left hand side of the array.

`answer:`

```
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 marks)*

A simple algorithm exists to compute the two's complement representation directly from the input binary number. It consists of two steps:

- Starting from the least significant bit, copy all the bits until the first 1 has been copied;
- After the first 1 has been copied, replace each of the remaining bits by their complement, i.e. replace all subsequent 1s by 0s and 0s by 1s.

Examples:

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

where $[N]_2$ denotes the two's complement representation of a number.

You must implement this algorithm using an **iterator** and the partial implementation of a list of bits described below and presented on the next two pages.

- Unlike most of the classes seen in class, the type of the elements that are stored in the list is **int**.
- Each element of a **BitList** is 0 or 1.
- The bits are stored in reverse order, i.e. the least significant bit is stored in the first node of the list. For instance, the list of bits 10011 would be represented by the following list:

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

- The empty list is equivalent to the list of bits that contains only 0.
- The instance method **iterator()** returns an object that implements the methods of the interface **Iterator**. Notice that the method **next()** returns an **int**, which is consistent with the type of the elements that are stored in the list.

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

You must complete the implementation of the instance method **BitList twosComplement()**. The method must be iterative (i.e. iterators must be used to traverse the list). The method returns a new instance of **BitList** and should not change *this* instance. Given an instance **b** that represents the list of bits 0101100,

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

the method call **b.twosComplement()** should return a new **BitList** instance that contains the following list of bits,

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

and **b** remains unchanged.

```
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'); // reverses the order
        }
    }

    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();
    }
    // an inner class to implement the iterator

    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 ; // move the cursor forward
            }
            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 marks)*

In this question there is a simple stack-based language to evaluate arithmetic expressions. The language for this question is actually a sub-set of a language called PostScript, which is a file format often used with printers. The main data-structure used by a PostScript interpreter is a stack. For the interpreter presented in the next pages, you must implement the operations **sub**, **exch** and **pstack**. Here are the 6 instructions of the language.

add: pops off the top two elements from the operands stack, adds them together and pushes back the result onto the stack.

sub: pops off the top two elements from the operands stack, subtracts them together and pushes back the result onto the stack. E.g.: $(3 - 1)$ would be represented as "`3 1 sub`".

eq: pops off the top two elements from the operands stack, compares them and pushes back the result (a boolean value) onto the stack.

exch: exchanges the order of the two elements on the top of the stack.

dup: duplicates the top element of the stack.

pstack: prints the content of the stack. It is important that the content of the stack remains the same after a call to the instruction pstack. The operation must not destroy or modify the content of the stack. Use the format of the example below.

The execution of the following PostScript program,

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

produces the following output,

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

The class **Calculator** presented on the next two pages is an interpreter for the language. The implementation requires three additional classes: **Stack**, **Token** and **Reader**. The class **Stack** implements the following three methods:

boolean isEmpty(); returns true if the stack contains no element.

Object pop(); removes and returns the element at the top of the stack.

Object push(Object element); pushes an element onto the top of the stack and returns the element.

```
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")) {

                // the implementation of the operation ‘‘add’’

            } else ... {

                // see next page

            }
        }
    }
}
```

The input for the method **execute** is a string that contains a valid PostScript program, e.g. “`1 pstack dup pstack`”. The input is parsed into *tokens* by the **Reader**. For the current example, the first call to the method **nextToken()** returns a token that represents the 1, the second call returns a token that represents the symbol `pstack`, and so on. The tokens that are not symbols are pushed onto the stack whilst the symbols are immediately evaluated. Here is a brief description of the classes Token and Reader.

Token: A token is an immutable object that represents either a boolean value, an integer or a symbol.

Reader: A reader parses the input string into Tokens.

On the next page, complete the implementation of the operations **sub**, **exch** and **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")) { // complete











            } 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")) { // complete











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











            } 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 marks)*

Write a **recursive** method that removes and returns the ith element of a singly linked list. The elements of the list are numbered from 0 to $(n-1)$. Therefore, considering an instance **l** that contains the elements, a, b, c and d, the method call **l.remove(2)** returns the element c and transforms the list so that the element c has been removed, and the content of the list after the call is a, b and d.

The method **public Object remove(int index)** is implemented following the approach presented in class, where a recursive method is made of a public part and a private recursive part, that we called the helper method. The public method is responsible for the modifications to the header of the list and initiates the first call to the recursive method. You must complete the definition of the helper method **Object remove(Node p, int index)** (see next page) for the singly linked list implementation below.

The method must throw the exception **IndexOutOfBoundsException** when appropriate. The method must not make any call to methods other than itself — you cannot assume the existence of any other method.

```
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;
    }

    // the rest of the implementation follows on the next page
    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
```

Appendix A: Boolean Algebra

Axioms and theorems of a Boolean Algebra. Given a set of values $\mathcal{B}$, that contains at least two elements, 0 and 1, and the following two binary operations, denoted $+$ and $\cdot$ (addition and product), and a unary operation, denoted by $\overline{a}$ (negation). We call $\mathcal{B}$ a Boolean algebra if the operations are closed on $\mathcal{B}$,

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

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

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

and the following axioms hold for all $a, b, c \in \mathcal{B}$.

Axiom 1 *commutativity:*

$$a + b = b + a$$

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

Axiom 2 *associativity:*

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

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

Axiom 3 *distributivity:*

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

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

Axiom 4 *identities, there are 2 elements,* $0, 1 \in \mathcal{B}$*, such that:*

$$a + 0 = a$$

$$a \cdot 1 = a$$

Axiom 5 *complementarity, for all* $a \in \mathcal{B}$*, there is an* $\overline{a} \in \mathcal{B}$ *such that:*

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

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

The following theorems can be demonstrated from the above axioms.

idempotence	$a \cdot a = a$	$a + a = a$
null elements	$a \cdot 0 = 0$	$a + 1 = 1$
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	$\overline{(\overline{a})} = a$	

Appendix B: TC-1101 assembly language

Mnemonic	opCode	Description
LDA	91	load x
STA	39	store x
CLA	08	clear (a=0, z=true, n=false)
INC	10	increment accumulator (modifies z and n)
ADD	99	add MDR to the accumulator (modifies z and n)
SUB	61	subtract MDR to the accumulator (modifies z and n)
JMP	15	unconditional branch to x
JZ	17	go to x if z==true
JN	19	go to x if n==true
DSP	01	display the content of the memory location x
HLT	64	halt

(blank space)