COS 126	General Computer Science	Spring 2013

Written Exam 1

This test has 8 questions, weighted as indicated. The exam is closed book, except that you are allowed to use a one-page single-sided cheatsheet. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided.

Print your name, login ID, and precept number on this page (now), and write out and sign the Honor Code pledge before turning in this paper. It is a violation of the Honor Code to discuss this exam until everyone in the class has taken the exam. You have 50 minutes to complete the test.

Write out and sign the Honor Code pledge before turning in the test:
"I pledge my honor that I have not violated the Honor Code during this examination."

Pledge: ____

____

Signature: ____________________

Name: ____________________

NetID: ____________________

Precept: ____________________

Problem	Score
1	/12
2	/10
3	/9
4	/9
5	/8
6	/9
7	/6
8	/7
Total	/70

P01	12:30 TTh	Dave Pritchard
P01A	12:30 TTh	Donna Gabai
P01B	12:30 TTh	Pawel Przytycki
P02	1:30 TTh	Tom Funkhouser
P02A	1:30 TTh	Allison Chaney
P02B	1:30 TTh	Pawel Przytycki
P02C	1:30 TTh	Vivek Pai
P02D	1:30 TTh	Siddhartha Chaudhuri
P03	2:30 TTh	Tom Funkhouser
P03A	2:30 TTh	Allison Chaney
P04	3:30 TTh	Vivek Pai
P04B	3:30 TTh	Shilpa Nadimpalli
P05	7:30 TTh	Shilpa Nadimpalli
P06	10am WF	Lennart Beringer
P07	1:30 WF	Dave Pritchard
P07A	1:30 WF	Kevin Lee
P07B	1:30 WF	Siyu Liu
P08	12:30 WF	Donna Gabai
P08A	12:30 WF	Judi Israel
P09	11am WF	Judi Israel

1. **Java Expressions** (12 points)

   For each of the Java expressions below, write down the type of the expression and its value. If the expression causes a syntax or run-time error, write an X in both boxes.

	Type	**Value**
`1 + 2 + "3" + 4 + 5`		
`(double)(1 / 2 + 1.0)`		
`false && (!(!!true || (true || !true)))`		
`7 = 11`		
`true != false`		
`Double.parseDouble("1E1")`		

2. **Number Systems** (10 points)

For this problem, we ask you to perform several calculations on hexadecimal numbers. For **each** part, we are using a **16-bit twos-complement** representation.

(a) What is `0ABE`, expressed in binary?

(b) What is `FFEE`, expressed in decimal?

(c) What is `F00D ^ FEED`, expressed in hexadecimal?

(d) What is `B0D1 & FACE`, expressed in hexadecimal?

(e) What is `B0D1 | FACE`, expressed in hexadecimal?

3. **Debugging** (9 points)

Recall that the absolute value function of x is defined by

$$\text{abs}(x) = \begin{cases} x, & \text{if } x \geq 0; \\ -x, & \text{otherwise.} \end{cases}$$

For example, $\text{abs}(4) = 4$ and $\text{abs}(-2) = 2$.

The following program is supposed to compute the sum of the absolute values of its arguments. Here is a sample run and the expected output:

```
% java AbsoluteSum 1 -2 4
The absolute sum is 7
```

However, your `AbsoluteSum` program is not working. Here is its source code:

```
1  public class AbsoluteSum {
2      public static void main(String[] args) {
3          int n = args.length;
4          int sum = 0;
5          for (int i = 0; i < n; n++) {
6              int value = Integer.parseInt(args(i));
7              if (value < 0);
8                  value = -1 * value;
9              sum = sum + value;
10         }
11         System.out.println("The absolute sum is " + sum);
12     }
13 }
```

For the three parts below, give the line number where there is a bug in the program, and a brief description of the bug. You do not need to write code to fix the bug.

(a) Find a syntax error that prevents the code from compiling.

Line: __________ Description: ____

(b) Find an error that causes the code to loop incorrectly (assuming the previous error was fixed).

Line: __________ Description: ____

After fixing these two bugs, you run the program and find it is computing the wrong value:

```
% java AbsoluteSum 1 -2 4
The absolute sum is -3
```

(c) Find the error that causes this incorrect output.

Line: __________ Description: ____

4. **Arrays** (9 points)

For this problem, you will trace the values stored in three arrays by the following program.

```
public class ThreeArrays {
    public static void main(String[] args) {
        int n = args.length;

        int[] a = new int[n];
        int[] b = new int[n+1];
        int[] c = b;

        for (int i = 0; i < n; i++)
            a[n-i-1] = Integer.parseInt(args[n-i-1]);

        for (int i = 0; i < n; i++)
            b[i+1] = b[i] + a[i];

        for (int i = 0; i < n; i++)
            c[i+1] = b[i] + c[i+1];
    }
}
```

If we run

```
% java ThreeArrays 1 10 100
```

what are the values stored in the arrays at the **end** of the program? Enter your responses in the boxes below.

a[0]:	a[1]:	a[2]:	
b[0]:	b[1]:	b[2]:	b[3]:
c[0]:	c[1]:	c[2]:	c[3]:

TOY Reference Card *Use this for the next problem on the facing page.*

```
             TOY REFERENCE CARD


INSTRUCTION FORMATS

             | . . . . | . . . . | . . . . | . . . .|
  Format 1:  | opcode  |    d    |    s    |    t   |  (0-6, A-B)
  Format 2:  | opcode  |    d    |       addr       |  (7-9, C-F)


ARITHMETIC and LOGICAL operations
    1: add              R[d] <- R[s] +  R[t]
    2: subtract         R[d] <- R[s] -  R[t]
    3: and              R[d] <- R[s] &  R[t]
    4: xor              R[d] <- R[s] ^  R[t]
    5: shift left       R[d] <- R[s] << R[t]
    6: shift right      R[d] <- R[s] >> R[t]

TRANSFER between registers and memory
    7: load address     R[d] <- addr
    8: load             R[d] <- mem[addr]
    9: store            mem[addr] <- R[d]
    A: load indirect    R[d] <- mem[R[t]]
    B: store indirect   mem[R[t]] <- R[d]

CONTROL
    0: halt             halt
    C: branch zero      if (R[d] == 0) pc <- addr
    D: branch positive  if (R[d] >  0) pc <- addr
    E: jump register    pc <- R[d]
    F: jump and link    R[d] <- pc; pc <- addr


Register 0 always reads 0.
Loads from mem[FF] come from stdin.
Stores to  mem[FF] go to stdout.
pc starts at 10

16-bit registers
16-bit memory locations
 8-bit program counter
```

5. **TOY** (8 points)

A NOOP (no operation) in a TOY program is *a command that has no effect*, other than that the program counter advances just past this command. One use of NOOPs is as a quick alternative to renumbering all of the lines in your TOY program, when you want to delete a line in the middle.

When we call a command a NOOP, we **cannot** make any assumptions about the state of the machine. For example, the command `1BB0` is a NOOP since it adds zero to register B, which cannot possibly have any effect on any register or memory location. But the command `1BBA` is **not** a NOOP because, depending on the contents of register A, this might change the value of register B.

Similarly, a *pair* of commands at memory locations L and $L+1$ forms a NOOP if reaching line L means that we are guaranteed to get to line $L+2$, with everything the same as it was at line L (except the program counter).

Determine which of the commands and pairs below are NOOPs. The ⋮ symbols represent hidden parts of the program. Do not make any assumptions about the hidden parts or the initial state of the machine. Circle your YES/NO answer for each of the 8 possible NOOPs.

Use the TOY reference card on the facing page.

⋮			
`20: D0D0`	This line is a NOOP:	YES	NO
⋮			
`30: BEEF`	This line is a NOOP:	YES	NO
⋮			
`40: 6991` `41: 5991`	This pair of lines is a NOOP:	YES	NO
⋮			
`50: 433E` `51: 43E3`	This pair of lines is a NOOP:	YES	NO
⋮			
`60: 2222`	This line is a NOOP:	YES	NO
⋮			
`70: 3333`	This line is a NOOP:	YES	NO
⋮			
`80: DA82` `81: CB82`	This pair of lines is a NOOP:	YES	NO
⋮			
`90: DA90` `91: CA91`	This pair of lines is a NOOP:	YES	NO
⋮			

6. **Methods and Input/Output** (9 points)

In this problem, you will analyze the program below:

```
public class Methodical {
    public static int transform(int x, int y) {
        x = x + 2;
        return (x + y);
    }
    public static int transform(double z) {
        int y = (int) z;
        StdOut.println(y);
        z = z + 1;
        return (int) z;
    }
    public static void main(String[] args) {
        String w = args[0];
        int x    = Integer.parseInt(StdIn.readString());
        int y    = Integer.parseInt(args[1]);
        double z = StdIn.readDouble();

        transform(z);
        StdOut.println(z);
        StdOut.println(w + transform(x, y));
    }
}
```

The file `numbers.txt` contains the following three lines:

4
5
6

(a) What is printed when we run `Methodical` with the arguments and input below?

`% java Methodical 1 2 3 < numbers.txt`

First line: ____________________

Second line: ____________________

Third line: ____________________

(b) What type of error occurs if we run this command?

`% java Methodical 1 2 3 < numbers.txt | java Methodical`

Circle one of I, II, III or IV.

I. No such element in `readString`
II. Array index out of bounds
III. Number format exception in `parseInt`
IV. Program runs forever

7. **Recursion** (6 points)

For the first four parts of this problem, you will investigate the behaviour of the recursive method defined by:

```
public static void f(int n) {
    // print n
    System.out.print(n + " "); // space to separate the outputs

    // recursive calls, but when n is zero, acts as the base case
    for (int i = 0; i < n; i++) {
        f(i);
    }
}
```

(a) What is printed when you call `f(0)`?

Output: ______________________________

(b) What is printed when you call `f(1)`?

Output: ______________________________

(c) What is printed when you call `f(2)`?

Output: ______________________________

(d) What is printed when you call `f(3)`?

Output: ______________________________

(e) For this part, we ask instead about the method `g`:

```
public static int g(int n) {
   if (n % 2 == 0) return n/10;
   return g(g(n/10));
}
```

What is the value of `g(3122013)`?

Value: ______________

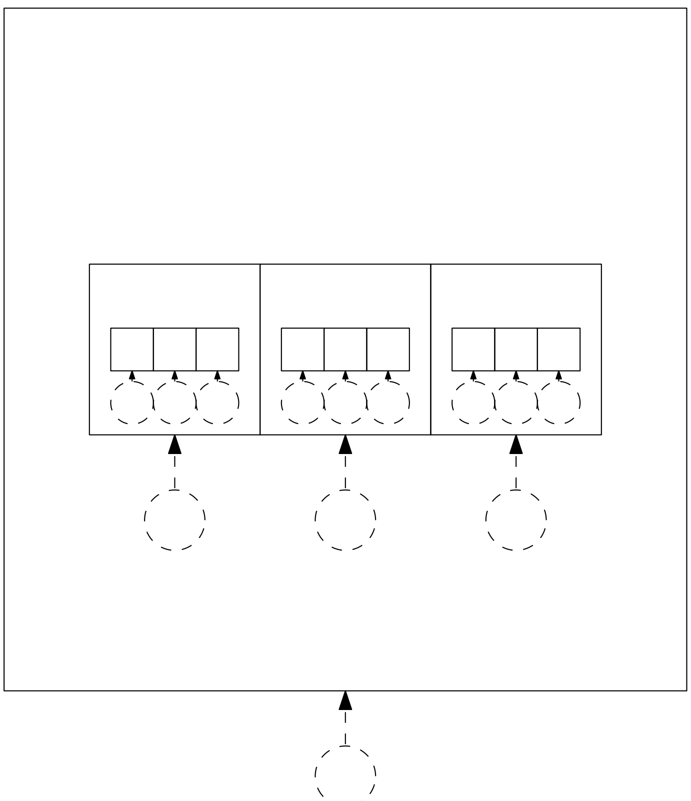
8. **Recursive Graphics** (7 points)

Here is a method that draws squares recursively:

```
public static void draw(int n, double x, double y, double r) {
    if (n==0) return; // base case

    draw(n-1, x, y, r/4);
    StdDraw.square(x, y, r);   // draw a square
    draw(n-1, x - r/2, y, r/4);
    draw(n-1, x + r/2, y, r/4);
}
```

Below, we plot the picture produced when `draw(3, 0.5, 0.5, 0.5)` is called. It draws thirteen squares, which we have also **labelled** with dashed circles and arrows.



(a) *What is the order in which the squares were drawn?* Write all of the integers from 1 to 13 in the circles to indicate this order, with 1 labelling the first square drawn and 13 the last.

(b) Which of the follow expressions represents the order of growth of the running time of `draw` as a function of the first argument n? Circle one.

- $\log_3 n$
- $n \log_3 n$
- n^3
- 3^n