COS 126	General Computer Science	Fall 2007

Exam 2

This test has 10 questions worth a total of 50 points. You have 120 minutes. The exam is closed book, except that you are allowed to use a one page cheatsheet, two sided, handwritten by you. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. Partial credit will be given for partially correct answers. **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."

_____

Signature

Problem	Score
0	
1	
2	
3	
4	
Sub 1	

Problem	Score
5	
6	
7	
8	
9	
Sub 2	

Total	

Name:

NetID:

Preceptor: Donna Adam Shirley JP Tom Ed Corey

The TOY reference card is on the last page of the exam.
Feel free to tear out the last page in order to use it more easily.

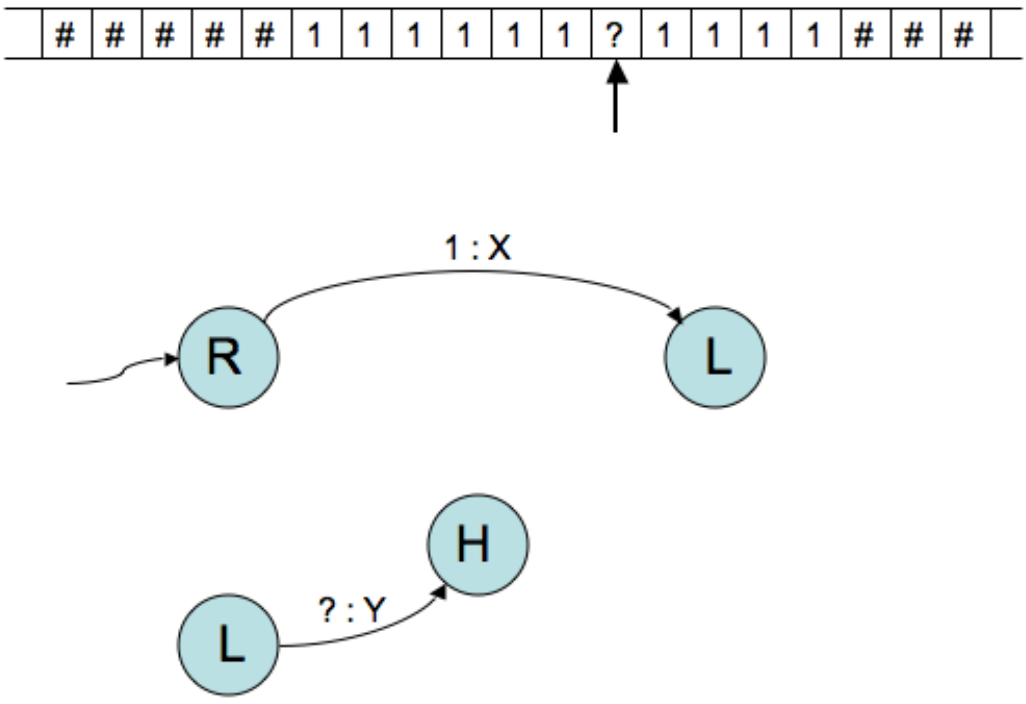
0. **Miscellaneous. (2 points) (really)**

   (a) Write your name and Princeton NetID in the space provided on the front of the exam, and circle the name of your preceptor.

   (b) *Write* and sign the honor code on the front of the exam.

1. **Turing Machines (5 points)**

   The Turing Machine below is supposed to decide if the unary number on the left of the question mark is greater than or equal to the unary number on the right, where the initial position of the tape head is at the question mark, as shown on the sample tape below. If it is, it prints "Y" on top of the question mark and then halts. If it isn't, it prints "N" and then halts. (The "unary" representation of an integer `n`, by the way, is just `n` consecutive 1's.) The Turing Machine's designer has quit unexpectedly, and you must finish the job. The designer's notes indicate that just one more state, and obviously a few state transitions, are needed to complete the machine. Please fill in the missing state and transitions.

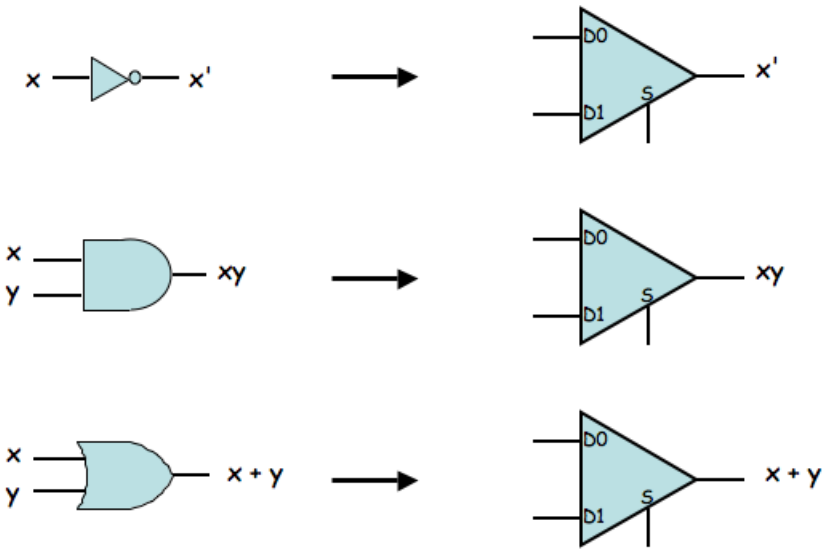


2. **Circuits (5 points)**

You know that with AND, OR, and NOT gates alone, you can express any Boolean function whatsoever—that the set of those gates is "universal." In this question you will show how a two-to-one mutiplexor is also universal by showing how each of AND, OR, and NOT can be made from one two-to-one multiplexor.

(a) First, make a truth table for a two-to-one multiplexor. This device has three inputs, which we'll label D0, D1, and S. The select input S controls which of D0 and D1 is copied to the output: when S is 0, the output equals D0, and when it's 1, the output equals D1.

(b) Show how to make each traditional logic gate from one multiplexor by setting each of the 3 inputs to x, y, 1, or 0.



3. **Audio list (8 points)**

In this question, you will finish an implementation of an Audio player. An AudioPlayer object holds the names of song files in a linked list. A client program can use the AudioPlayer to build the list and play songs by calling methods from the following API:

```
public class AudioPlayer
--------------------------------------------------------------------------------
         AudioPlayer()                  // create empty list of song filenames
   void addSong(String filename)        // add one song filename to the end of the list
   void playAll()                       // plays all songs on list, prints each title
   void skipToThisSong(String filename)  // make filename song current, play and print
```

Please implement the three methods listed above and part of the `main` method in the boxes provided below. You'll see that `main` reads from StdIn a series of song filenames separated by white space. Use the String method `equals()` to compare song filenames. (If `a` and `b` are Strings, then `a.equals(b)` returns `true` when `a` and `b` are equal.) Feel free to call one method from another method.

Here's a test run of AudioPlayer (without the sound!):

```
% more songs.txt
AskMeWhy.mid ISawHerStandingThere.mid TwistAndShout.mid
% java AudioPlayer < songs.txt
Now Playing: ISawHerStandingThere.mid
```

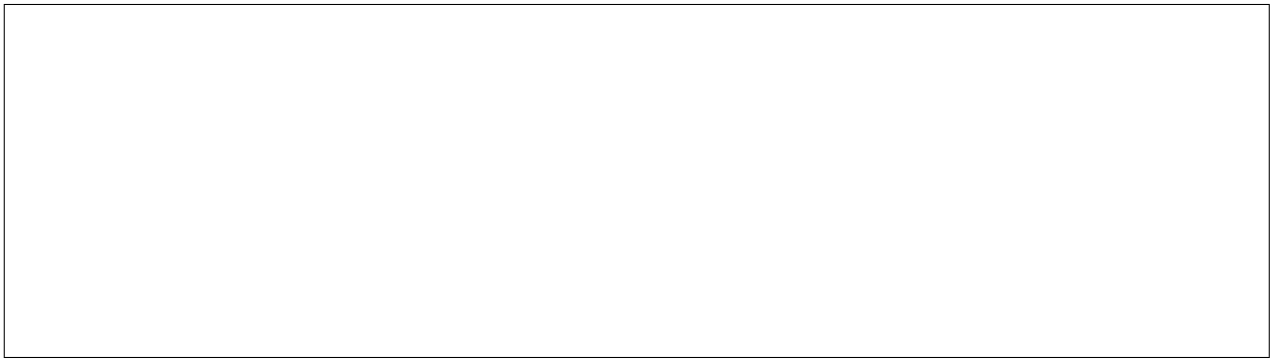
```
public class AudioPlayer {
    private Node start;  // first song
    private Node end;    // last song
    private Node cur;    // current song

    private class Node {
        private String filename;
        private Node next;
    }

    public AudioPlayer() {
        start = null; end = null; cur = null;
    }

    // add song to the end of list
    public void addSong(String filename) {
```

```
    }
```

```
    // play and print the current song
    private void playCur() {
        System.out.println("Now Playing: " + cur.filename);
        StdAudio.play(cur.filename);
    }

    // play and print all songs
    public void playAll() {
```

```
    }

    // make requested song current, play it and print it
    // assume it is in the list
    public void skipToThisSong(String filename) {
```

```
    }

    // main (test client)
    public static void main(String[] args) {
        AudioPlayer player = new AudioPlayer();
        // read song filenames from StdIn and store in the
        // linked list
```



```
        player.skipToThisSong("ISawHerStandingThere.mid");
    }
}
```

4. **TOY (6 points)**

TOY programmers, like most programmers, would often like to use a stack. Below is a program with space set aside for data, the stack, and for the code for push and pop. Push has already been implemented on lines 50-56. Your job is to finish implementing pop using lines 61-66.

The TOY client code that calls the pop and push functions uses the TOY jump-and-link instruction, which will save the caller's PC in Register F. FF50 will thus be a call of the push function, and FF60 will call pop. Both functions use TOY's jump-register instruction EF00 to return to the caller. The stack starts at D0 and grows toward larger addresses. You don't need to implement any code to check if the stack is empty or full, nor any client code.

To push, the client will first have put the 16-bit value to be pushed in memory location 03, and will expect pop to have put the popped value there when it returns.

The TOY reference card is on the last page of the exam.

```
// data
02: 00D0                          stack pointer
03: 0000                          item to push or pop (client supplies or consumes)

//push
50: 7101   R[1] <- 1              constant 1
51: 8202   R[2] <- mem[02]        stack pointer to R[2]
52: 8303   R[3] <- mem[03]        item to push gets loaded into R[3]
53: B302   mem[R[2]] <- R[3]      item gets pushed onto stack
54: 1221   R[2] <- R[2] + R[1]    increment stack pointer
55: 9202   mem[02] <- R[2]        store new stack pointer
56: EF00   goto R[F]              TOY's return statement

//pop
60: 7101   R[1] <- 1              constant 1

61:

62:

63:

64:

65:

66:


//stack
D0: DEAD                          first item pushed will go here
D1: BEEF                          (stack grows down)
D2: DEAD                                        |
D3: BEEF                                        |
D4: ...                                         V
```

5. **Object Oriented Programming (4 points)**

You're given code for a class, STLite, which is a simplified version of the ST (symbol table) class used in the programming assignments, differing only in that the key is an int (which you can assume is positive) and the value is a String. Here's the API for STLite:

```
public class STLite
--------------------------------------------------------------------------------
           STLite()                     /* construct a new STLite */
      void put(int key, String value)   /* create an entry for the given key,
                                           with the given value */
    String get(int key)                 /* return the value associated with the
                                           given key, or null if there is no entry
                                           for that key */
```

You want to make a new class, STNew, which has the same API as STLite but is implemented differently. Specifically, each STNew object should contain ten STLite objects, corresponding to the final decimal digit of the key. You will want to create an array of STLite objects in your STNew constructor.

Write code for the STNew class, which should use STLite.

```
public class STNew {




     public STNew() {




     }

     public void put(int key, String value) {




     }

     public String get(int key) {




     }
}
```

6. **Queues (7 points)**

Here is the API for the Queue datatype:

```
public class Queue<Item>
-----------------------------------------------------------
             Queue()       // create an empty Queue
    boolean  isEmpty()
       void  enqueue(Item item)
       Item  dequeue()
       Item  peek()        // next Item to be dequeued
```

Write a static client method `QueueMerge()` that take two queues of integers whose elements have already been sorted in ascending order (first to be dequeued is smallest), and which will merge those two queues, making a third queue with all the integers of both original queues, in ascending order. The original queues need not be saved.

```
public static Queue<Integer> QueueMerge(Queue<Integer> r, Queue<Integer> s)
{









}
```

7. **Regular Expressions (6 points)**

(a) How many unique strings are described by each of the following regular expressions? (The symbol + means "one or more".)

(i) (A|B)(A|B) Answer:

(ii) AB+ Answer:

(iii) AB* Answer:

iv) AB Answer:

(b) Which of the following regular expressions specifies a set of strings that can be accepted by some deterministic finite state automaton? Circle the ones that can.

(i) (A|B)(A|B)

(ii) AB+

(iii) AB*

(iv) AB

(c) In Java, the regular expression "\d" matches any digit. Write an equivalent expression without using a backslash.

(d) Here is a list of strings:

A: `alphabet`
B: `abracadabra`
C: `babcock`
D: `hubbub`
E: `suburbia`
F: `dabchick`

Write the letters corresponding to all of the words that contain the following regular expressions. As an example, the answer to (i) is already given. (The symbol . (dot) is a wildcard that indicates one occurrence of any character.)

(i) `ab` Answer: ABCF

(ii) `abc` Answer:

(iii) `a.*a` Answer:

(iv) `........*` (eight dots) Answer:

(v) `bu(b|r)` Answer:

8. **Theory: True or False (5 points)**

   Write T for true or F for false next to each of the following statements, according to their veracity.

   _____ The Church-Turing Thesis is called a thesis and not a theorem because it is a statement about the real world that cannot be formally proved.

   _____ It's possible to write a program that can decide whether another program solves the halting problem.

   _____ In general, it is undecidable whether a Turing Machine will halt on a given input.

   _____ In general, it is undecidable whether a Turing Machine will halt on a given input after at most n steps.

   _____ If a problem is in P, then any program that solves that problem must run in polynomial time.

   _____ If a problem is in P, then it's possible to write a program that checks proposed solutions to that problem in polynomial time.

   _____ If a problem is in NP, then it's possible to write a program that checks proposed solutions to that problem in polynomial time.

   _____ If P=NP, then the Traveling Salesperson Problem can be solved in polynomial time.

   _____ If the Traveling Salesperson Problem can be solved in polynomial time, then P=NP.

   _____ When a DFA is processing a particular input string, its running time will always be polynomial in the length of that input string.

9. **Nugget (2 points)**

   What is one intriguing computer science factoid or idea from this course that you shared with a parent or a friend at some point, or that you think you might? (2 points for any serious answer)

```
 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
```