Computer Science 126 — 01/21/2000
Final Exam with Answers — 1:30 - 4:30pm

1. Linked Lists and Recursion [11]

This is based on final review question #6.

Assume the following linked list definition.

```
typedef struct node *link;
struct node {
    int account;
    float amount;
    link next;
};
```

a) Write a function that returns a new list with a new node added to the beginning of a given list. The prototype is given below. The new node is to contain **`account`** and **`amount`** in it.

```
link
insertHead (link list, int account, float amount)
{
    link t = (link) malloc(sizeof *t);
    t->next = list;
    t->account = account;
    t->amount = amount;
    return t;
}
```

b) Complete the following recursive function which returns the sum of the **`amount`** values in the list:

```
float
totalAmount(link list)
{
    if (list == NULL)

        return 0.0;                                    ;

    return list->amount + totalAmount(list->next)      ;
}
```

c) What does the following program fragment print?

```
link
foo(link list, int a)
{
    if (list == NULL)
        return NULL;
    if (list->account <= a) {
        list->next = foo(list->next, a);
        if (list->account == a)
            return list->next;
    }
    return list;
}

/* just prints the list */
void
print(link list)
{
    link t;
    for (t = list; t != NULL; t = t->next)
        printf("[%d %f] ", t->account, t->amount);
    printf("\n");
}

int
main()
{
    link list = NULL;
    list = insertHead(list, 4000, 50.0);
    list = insertHead(list, 3000, 20.0);
    list = insertHead(list, 2000, 70.0);
    list = insertHead(list, 1000, 40.0);
    print(list);
    print(foo(list, 2500));
    print(foo(list, 3000));
}
```

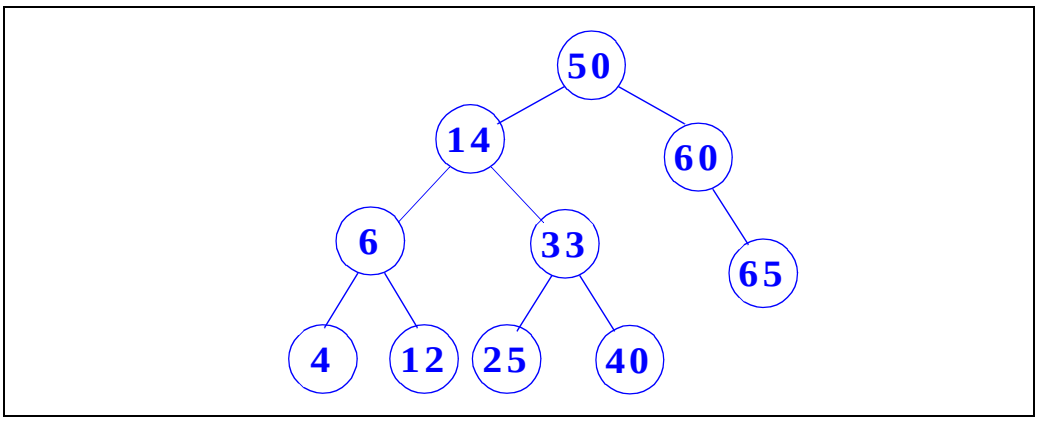
```
[1000 40.000000] [2000 70.000000] [3000 20.000000] [4000 50.000000]
[1000 40.000000] [2000 70.000000] [3000 20.000000] [4000 50.000000]
[1000 40.000000] [2000 70.000000] [4000 50.000000]
```

The four calls to **`insertHead()`** make a sorted linked list. **`foo()`** is a function that deletes a node with the given **`account`** field. It takes advantage of the fact that the list is sorted so it stops searching as soon as it finds that the current list element is bigger than the desired element.

2. Binary Search Trees [9]

a) Insert the following keys into an empty binary search tree in the given order. Draw the resulting tree.

50, 14, 60, 65, 33, 25, 6, 4, 12, 40



b) Print the keys in the above tree using postorder traversal.

```
4, 12, 6, 25, 40, 33, 14, 65, 60, 50
```

c) For a **general binary search tree** with **more than two nodes** of **distinct keys** (not necessarily the tree above), answer the following questions.

c.1) Under what circumstances, if any, can the preorder and postorder traversals produce the same result?

Under no circumstance. (Think about where the root is printed.)

c.2) Under what circumstances, if any, can the inorder and postorder traversals produce the same result?

A degenerated tree with no right branches anywhere.

c.3) Under what circumstances, if any, can the preorder and inorder traversals produce the same result?

A degenerated tree with no left branches anywhere.

3. Arrays, Pointers, and Complexity [10]

```
void
g(int a[], int asize)
{
    int *p, *q;
    int t;
    for (p = &a[asize-1]; p >= &a[0]; p--) {
        for (q = &a[0]; q < p; q++) {
            if (*q > *(q+1)) {
                t = *q;
                *q = *(q+1);
                *(q+1) = t;
            }
        }
        printf("%d ", *p);
    }
}

main()
{
    int a[8];
    int i;
    for (i = 0; i < 8; i++) a[i] = 8-i;
    g(a, 8);
}
```

This is based on final review question #20.

a) What does this program print?

`8 7 6 5 4 3 2 1`

The function **`g()`** sorts the given array. Each iteration of the outer loop puts the largest unsorted element at the right place. The **`printf()`** statement prints this element.

b) What is the complexity of the function **`g()`** in "big O" notation?

The comparison statement is executed [asize*(asize-1)/2] times. So the complexity is $O(n^2)$, where n is asize.

c.1) Is this function in the complexity class P?

Yes. $O(n^2)$ is polynomial time.

c.2) Is this function in the complexity class NP?

Yes. P is a subset of NP.

c.3) Is this function in the complexity class NP-Complete?

No. No polynomial time algorithm has been discovered for any NP-Complete problem.

4. Java [8]

This is based on final review question #34 and #33.

What does the following program print?

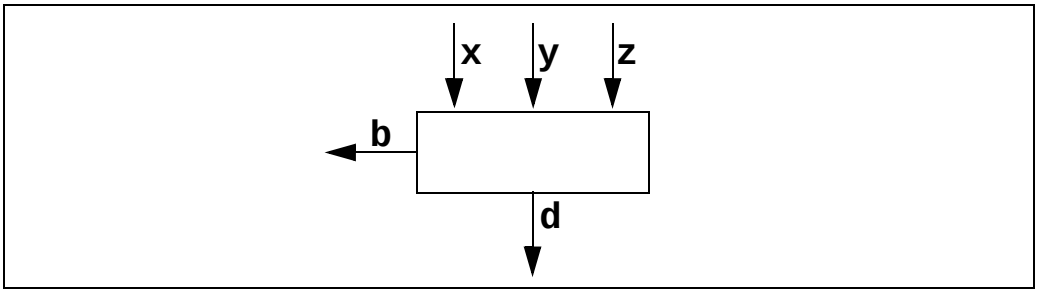
```
public class A {
    int x,y;
    public A() {x = 1; y = 2;}
    public void change(int x) {this.x = x;}
    public int foo() {return x*y;}
    public int bar() {return x;}
}

public class B extends A {
    int x = 5;
    public int foo() {return x+y;}
    public int baz() {return x;}
}

class Test {
    public static void main(String[] args) {
        B b = new B();
        B x = b;
        System.out.println(b.foo());
        b.change(10);
        System.out.println(b.foo());

        System.out.println(b.bar());
        System.out.println(b.baz());

        System.out.println(x.bar());
        System.out.println(x.baz());
    }
}
```

```
7
7
10
5
10
5
```

There are actually two different variables named **x**. All the methods of class **B** access the one within the scope of class **B**. All the methods of class **A** access the one within the scope of class **A**. Furthermore, all Java objects are references, which are basically pointers. So changing **b** automatically changes **x**. If you want to make a copy, there is an explicit Java method to invoke to accomplish this.

5. Combinational Circuits and Two's Complement Arithmetic [11]

We first construct a one-bit (bit-slice) subtracter. Its interface is as the following:



It has three inputs and two outputs: **x** is the bit to subtract from; **y** is the bit to subtract; **z** is the input borrow bit that is also subtracted from **x**; **d** is the output difference bit; and **b** is the output borrow bit.

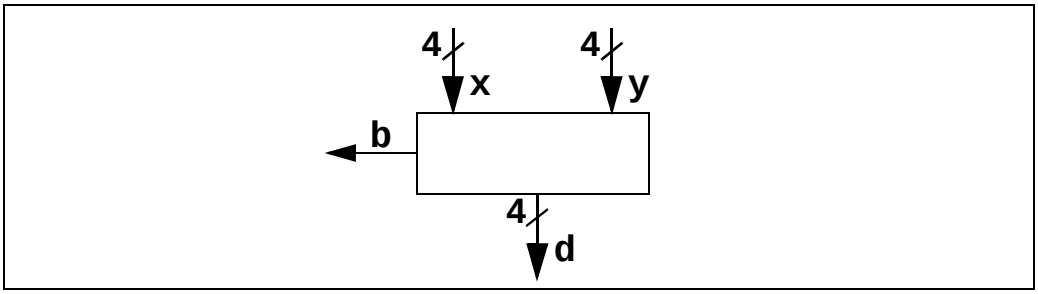
a) Derive all the necessary truth tables.

xyz	db
000	00
001	11
010	11
011	01
100	10
101	00
110	00
111	11

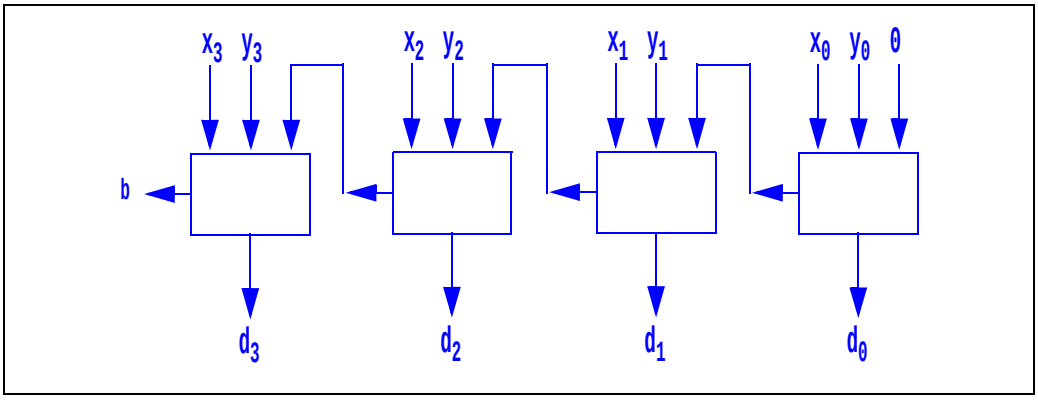
b) Derive all the output boolean expressions.

```
d = x'y'z + x'yz' + xy'z' + xyz
b = x'y'z + x'yz' + x'yz + xyz = x'y + x'z + yz
```

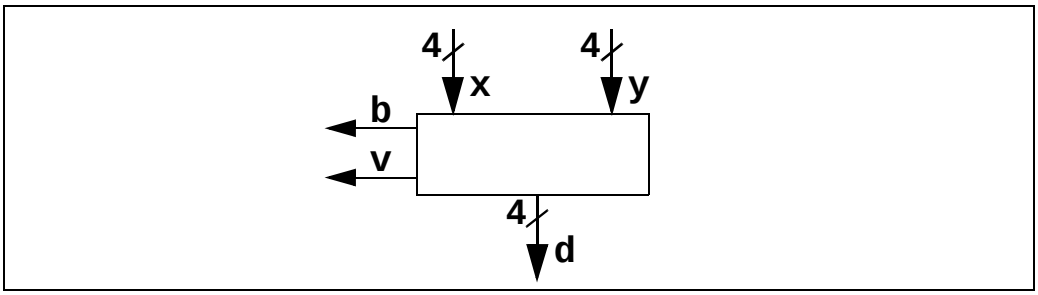
c) We now want to construct a 4-bit subtracter that has the following interface:



Recall that the notation used in this illustration means that **x** is a 4-bit wire $x_3x_2x_1x_0$. Similarly, $y=y_3y_2y_1y_0$ and $d=d_3d_2d_1d_0$. Illustrate how to use the bit-slice subtracter of the previous page to build a 4-bit subtracter.



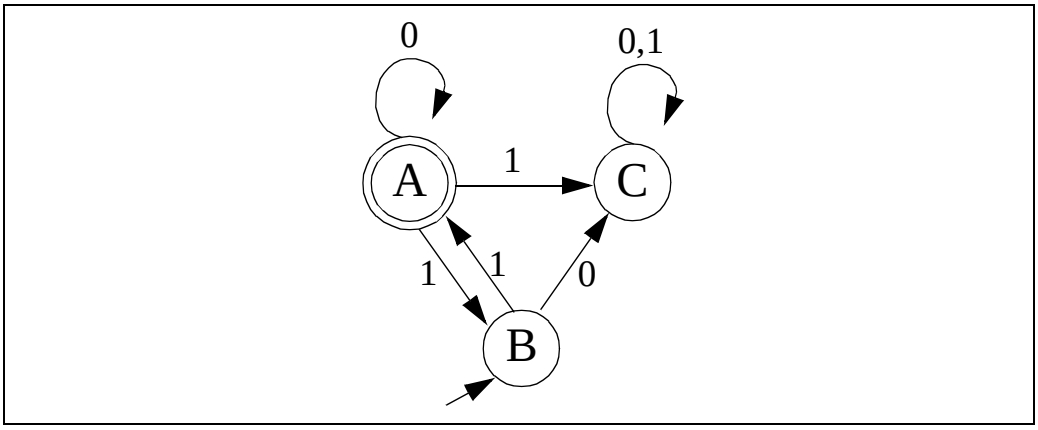
d) We now want to add an overflow output bit **v** to the subtracter (The inputs **x** and **y** are two two's complement integers):



Recall that the overflow bit is 1 iff the answer of the subtracter does not fit in 4 bits. Are the borrow bit **b** and the overflow bit **v** the same?

No. Consider the example when `x=0` and `y=-1`. I will leave the derivation of a boolean expression for the overflow bit as an exercise (for future classes). (Hint: you only need to consider the left most bits of the input and some of the output bits.)

6. FSAs and Grammar [11]



In this non-deterministic FSA, **B** is the start state, and **A** is the accept state.

This is loosely based on final review question #14.

a) Give the Type 3 grammar that corresponds to this machine.

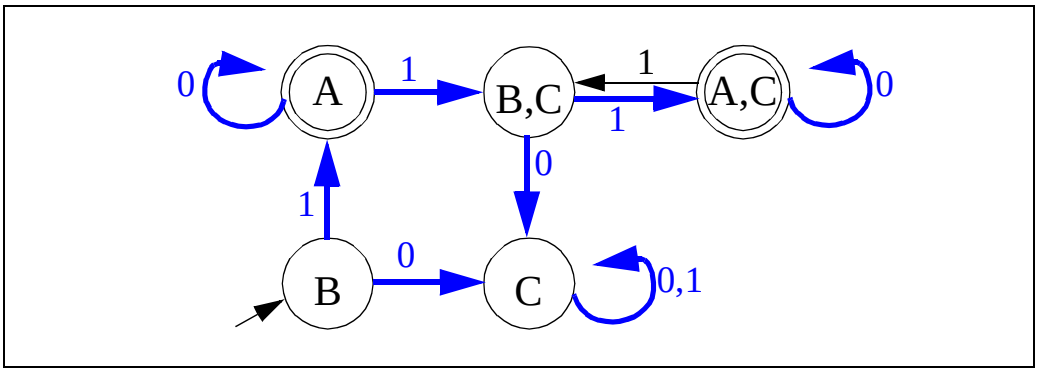
B -> ε
A -> B1
A -> A0
B -> A1

ε is the empty string.

C is a dead sink state. The remaining rules concerning state C are not needed but ok to include:

C -> A1
C -> C1
C -> C0
C -> B0

b) Convert this non-deterministic FSA into a deterministic FSA by filling in the missing transition edges in the following graph.



c) Which one of the following regular expressions does **not** describe this FSA?

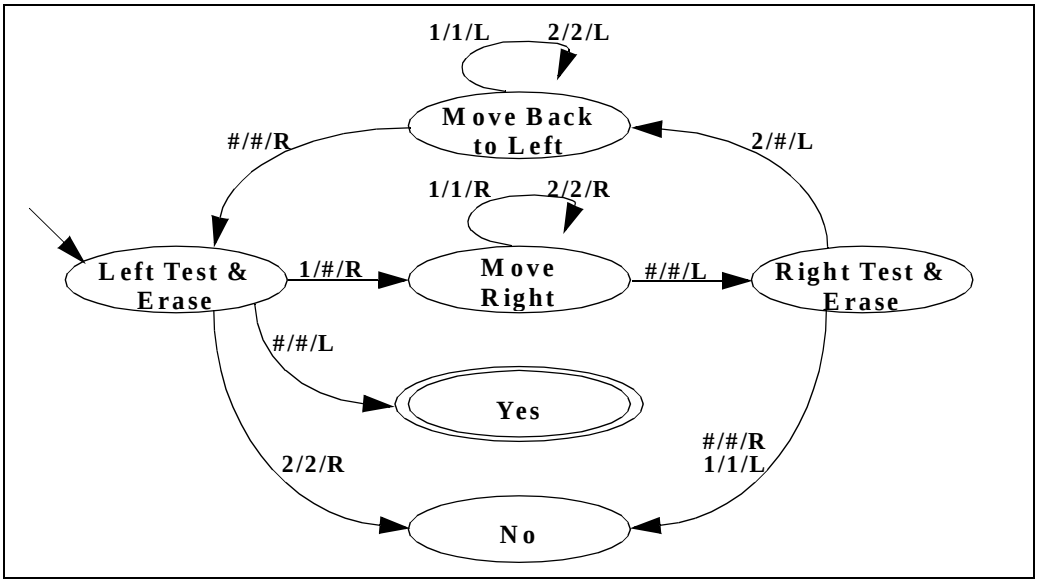
A) 10*(110*)*
B) (10*)|(10*110*(110*)*)
C) (10*)|(10*11(0*11)*)

C)

A can be derived from looking at the non-deterministic FSA. B can be derived from the deterministic FSA. It's also easy to see that A and B are equivalent.

C demands that if the sequence 11 shows up in the expression, it must end with 11, an unnecessary requirement.

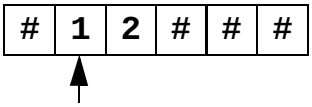
7. Automata [11]



This is loosely based on the Parlindrome Turing machine in the course packet.

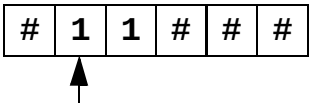
In the Turing machine illustrated above, The state labeled with "Left Test & Erase" is the start state; and "Yes" is the accept state.

a.1) Does the machine accept the following string (given the initial read head position)?



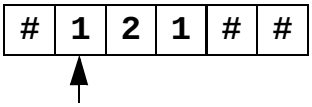
Yes.

a.2) Answer the same question for the following string.



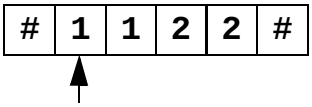
No.

a.3) Answer the same question for the following string.



No.

a.4) Answer the same question for the following string.



Yes.

b) Characterize the strings that are accepted by this Turing machine.

The way the Turing machine works is that its head bounces back and forth between the two ends of the input string, matching 1s on the left end with 2s on the right end and erasing them. It halts in the Yes state when the string becomes empty.

So the strings it accepts are of the form 1^k2^k.

c) Does there exist a push down automaton that can recognize the language accepted by this Turing machine?

Yes. (See “extra language exercises”, PDA exercise #1.)

d) Does there exist a finite state automaton that can recognize the language accepted by this Turing machine?

No. (FSAs can’t count so it can’t match the number of 1s with that of 2s.)

8. TOY Architecture [12]

This is based on the hardware exercises #17-20 and #9 of the second midterm.

Assume that each of the following operations in the TOY datapath takes the stated amount of time (in nanoseconds):

	Operation	Time (ns)
1	**fetching an instruction from memory**	**0.8**
2	**reading from the register file**	**0.8**
3	**executing any ALU operation**	**0.8**
4	**accessing data memory**	**0.9**
5	**writing the result back into the register file**	**0.8**
6	**all other delays (those of MUXes, controls, wires, etc.)**	**0**

(Each of these operations must finish within a single clock cycle.)

a) How much time (in nanoseconds) does each of the following instructions **minimally need** to complete?

a.1) **`9923  r1 <- mem[r2+r3]`**

instruction fetch + register file read + add + memory load + register file write
= 0.8 + 0.8 + 0.8 + 0.9 + 0.8
= 4.1 ns

a.2) **`1114  r1 <- r1+r4`**

instruction fetch + register file read + add + register file write
= 0.8 + 0.8 + 0.8 + 0.8
= 3.2 ns

a.3) **`A923  mem[r2+r3] <- r1`**

instruction fetch + register file read + add + memory store
= 0.8 + 0.8 + 0.8 + 0.9
= 3.3 ns

b)

b.1) What is the minimum cycle time for a single cycle design?

The longest instruction delay, namely, that of the load instruction, 4.1 ns.

b.2) What is the minimum cycle time for a multicycle design?

The slowest operation in the table, namely, the memory access, 0.9 ns.

c) Given the following TOY program:

```
1A: 9923    r1 <- mem[r2+r3]
1B: 1114    r1 <- r1+r4
1C: 1114    r1 <- r1+r4
......                          } 8 times
22: 1114    r1 <- r1+r4
23: A923    mem[r2+r3] <- r1
```

c.1) Assume a single cycle design and the cycle time of b.1), how much time does this program **actually take**?

There are a total of 10 instructions in this program, each of those takes the cycle time of 4.1 ns, so the total running time is just 4.1*10 = 41 ns.

c.2) Assume a multicycle design and the cycle time of b.2), how much time does this program **actually take**?

The cycle time is 0.9 ns. The memory load instruction takes 5 cycles. The add and store instructions take 4 cycles each and there are 9 of them. So the total running time is 0.9 * (5 + 4*9) = 36.9 ns.

c.3) Using the same assumptions, is one of these two designs **always** faster than the other for **any** program?

No.

For this particular program, the multicycle design is faster. But notice that the memory load instruction actually takes longer so if there are a lot of those, the multicycle design is slower.

9. ADT, TOY, Compilers, Two's Complement Arithmetic, Operating Systems [17]

This is loosely and partially based on the final review exercises #27 and #29 and #10 of the second midterm.

Consider the following arithmetic expression:

`(1*2)-(3*(9-2))`

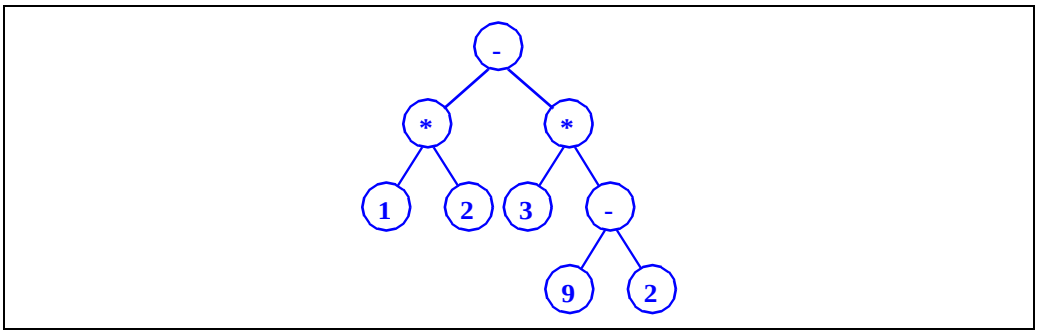
a) Give the postfix representation of this expression.

`12*392-*-`

b) Evaluate this postfix expression using a stack. Give the content of the stack when the stack is tallest. Graphically point out the top of the stack.

After the first * operator is evaluated, a 2 is the only thing in the stack. Right before the - operator is evaluated, the stack reaches its tallest height. The content from bottom to top is "2, 3, 9, 2".

c) Give the syntax tree that our TOY compiler constructs for this expression.



d) Fill in the missing pieces of the machine code generated by our TOY compiler for this expression.

```
1A: B101        R1 <- 1
1B: B202        R2 <- 2
1C: 3112        R1 <- R1 * R2
1D: B203        R2 <- 3
1E: B309        R3 <- 9
1F: B402        R4 <- 2
20: 2334        R3 <- R3 - R4
21: 3223        R2 <- R2 * R3
22: 2112        R1 <- R1 - R2
23: 4102        print R1
24: 0000        halt
```

e) Give the content of **R1** in hexadecimal when TOY halts after executing the above program.

FFED

f) Change the arithmetic expression to the following c statement

```
a = (a*2)-(3*(9-2))
```

where **a** is an integer variable that is stored at memory location **0x00**. Assume **R0** holds the value 0. What changes should the TOY compiler make to the machine code of part d)?

Change

```
1A: B101        R1 <- 1
```

to

```
1A: 9100        R1 <- mem[0x00]
```

and add a store instruction after all is computed:

```
23: A100        mem[0x00] <- R1
```

g) We decided to add multiprogramming support to TOY. To do this, 1) we added timer interrupts to TOY, and 2) we wrote a small TOY Operating System that takes over control when a timer interrupt occurs. Of the two above TOY programs [program in part d) and program in part f)], can we run multiple instances of them simultaneously?

A) Program d only
B) Program f only
C) Both
D) Neither

A)

We can not run multiple instances of the program in f) because it modifies physical memory and multiple active instances may interfere with each others' operation.

Appendix. TOY Instruction Set

INSTRUCTION FORMATS	
Format 1: opcode, r0, r1, and r2 Format 2: opcode, r0, and 8-bit addr Indexed addressing (for format 2): if leading bit of r0 digit is 1, then addr = r1 + r2	
TRANSFER between registers and memory	
9: load A: store B: load address	r0 <- mem[addr] mem[addr] <- r0 r0 <- addr
ARITHMETIC operations	
1: add 2: subtract 3: multiply	r0 <- r1 + r2 r0 <- r1 - r2 r0 <- r1 * r2
LOGICAL operations	
C: xor D: and E: shift right F: shift left	r0 <- r1 ^ r2 r0 <- r1 & r2 r0 <- r0 >> addr r0 <- r0 << addr
CONTROL	

0: halt
4: system call
5: jump
6: jump if positive
7: jump and count
 	
8: jump and link
	halt
	print r0 on tty
	pc <- addr
	if (r0 > 0) pc <- addr
	r0--
	if (r0 != 0) pc <- addr
	r0 <- pc
	pc <- addr

Feel free to tear out this sheet.