1. (Finite State Machine Design)

(a) (7 points) We are given that all prefixes satisfy $n0 \leq n1 \leq n0 + 2$. Let $d = n1 - n0$. Then we have $0 \leq d \leq 2$. For the machine in Figure 1 each state is labeled with a value of d: 0, 1, 2 or other.

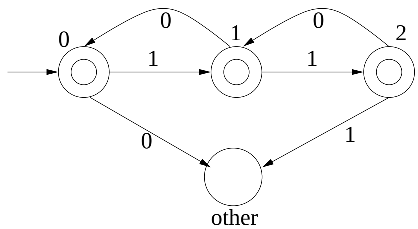

Figure 1: Solution to Problem 1a

(b) See Figure 2; each state is labeled with the remainder of division by 3: 0, 1, or 2.

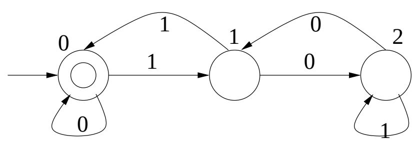

Figure 2: Solution to Problem 1b

(c) See Figure 3. The left state is entered initially and following the first white space in a block of white spaces. The right state is entered when a non-white space is seen. Here `-` denotes a white space and `a` any other symbol.

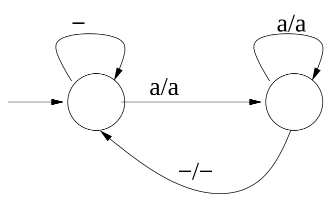

Figure 3: Solution to Problem 1c

(d) See Figure 4. States 1 and 2 define the behavior of the machine when it is operating on an expression which is not within a parentheses;

states 3 and 4 are the corresponding states when a "(" has been seen. The machine is in state 1 initially and whenever a digit is expected. And it is in state 2 when an operator is expected. The meanings of states 3 and 4 are analogous.

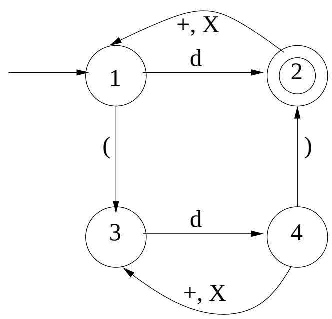

Figure 4: Solution to Problem 1d

2. (Finite State Machine Theory)

   (a) Let t denote the string associated with a state. We first postulate a predicate with each state. Predicate $A(t)$ holds for every string t that leads to state A; similarly for the other states. See Figure 5.

   $A(t)$:: t is a repetition of 1s (possibly empty).
   $B(t)$:: t is of the form $x0$, and 011 is not a substring of x.
   $C(t)$:: t is of the form $x01$, and 011 is not a substring of x.
   $D(t)$:: 011 is a substring of t.

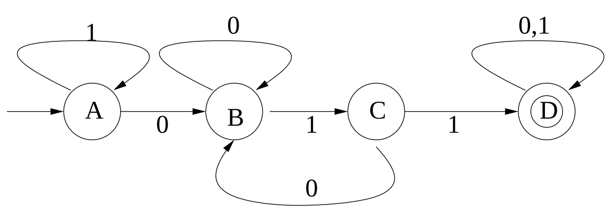

Figure 5: Transducer in Problem 2

   The theorems to be proven are:

   $A(\epsilon)$, $A(t) \Rightarrow A(t1)$, $A(t) \Rightarrow B(t0)$,
   $B(t) \Rightarrow B(t0)$, $B(t) \Rightarrow C(t1)$, $C(t) \Rightarrow B(t0)$,
   $C(t) \Rightarrow D(t1)$, $D(t) \Rightarrow D(t0)$, $D(t) \Rightarrow D(t1)$

   (b) The strings of the language defined by the regular expression $(a \mid ab)(c \mid bc)$ are ac, abc and $abbc$. Another regular expression, using both alternation and concatenation, that denotes the same language is $a(\epsilon|b|bb)c$.

(c) i. Each string (in the language) has at least one 1: $0^*1(0|1)^*$

ii. Each string has at most one 1: $0^*|0^*10^*$

iii. Each string has exactly one 1: 0^*10^*

iv. Every block of 1s in a string is of even length: $(0^*(11)^*)^*$

3. (Functional Programming)

(a)

```
xor 0 0 = 0
xor 0 1 = 1
xor 1 0 = 1
xor 1 1 = 0
xor x y
 | (even x) && (even y) = 2 * (xor p q)
 | (even x) && (odd y)  = 2 * (xor p q)+1
 | (odd x)  && (even y) = 2 * (xor p q)+1
 | (odd x)  && (odd y)  = 2 * (xor p q)
                          where
                           p = x `div` 2
                           q = y `div` 2
```

Note: The first 4 lines may be replaced by using the following conditional equation:

```
| x == 0  && y == 0 = 0
```

(b) We first compute $h(n) = (g(2n), g(2n+1))$, using a scheme similar to that for `fibpair`, given in the class notes.

```
h 0 = (0,1)
h n = (x+y,x)
      where
       (x,y) = h(n-1)
```

Then,

```
g(m)
 | even m = fst(h(m `div` 2))
 | odd m  = snd(h(m `div` 2))
```

(c)

```
g 0 = (f 0 == 0)
g (n+1) = (g n) && (f(n+1) == 0)
```

(d)

```
power2new 0 = 1
power2new n
 | even n = r * r
 | odd n  = r * r * 2
            where r = power2new (n `div` 2)
```