1. [20] The function **count** has two arguments, the second of which is a list. It returns the number of times the first argument occurs as an element of the second. For example,

(count 'e '(G r e e c e)) returns 3

Give Scheme code for three different implementations of this function:

countA is a low-level implementation using recursion, but no helper functions.

countB is a low-level tail-recursive implementation.

countC uses common higher-order functions, and possibly a lambda expression.

[5 points]:

```
(define (countA x L)
  (if (null? L)
      0
      (if (equal? x (first L))
          (+ 1 (countA x (rest L)))
          (countA x (rest L)))))
```

[5 points]:

```
(define (countB x L)
  (define (countBtr x L acc)
    (if (null? L)
        acc
        (if (equal? x (first L))
            (countBtr x (rest L) (+ 1 acc))
            (countBtr x (rest L) acc))))
  (countBtr x L 0))
```

[5 points]:

```
(define (countC x L)
  (length (filter (lambda(y) (equal? x y)) L)))
```

[5 points]: Contrast the memory usage for these three versions.

countB uses the **least** amount of memory, because it is tail recursive. It uses a **constant** amount of memory.

countA uses extra memory for the **recursion stack**. The amount is proportional to the length of the list A.

countC creates an **intermediate list** of length equal to the answer. Depending on how filter and length are implemented, it could also use memory for the recursion stack.

---

2. [20] Construct using Scheme a function **cluster** that clusters names associated with cities. The argument is a list of pairs, such as:

```
'((Adams SanFrancisco)
  (Jefferson LosAngeles)
  (Thomas SanFrancisco)
  (Lincoln LosAngeles)
  (Lewis Seattle)
  (Clark LosAngeles)
  (Washington Seattle))
```

The result of *cluster* is an association list that lists each city once, followed by the names associated with the city. For the above example, it might be:

```
'((LosAngeles Jefferson Lincoln Clark)
  (SanFrancisco Adams Thomas)
  (Seattle Lewis Washington))
```

Using some higher-order functions is recommended.

[20 points]: This is one of many ways: Get the list of cities alone, by mapping second over the original list, then removing duplicates. Map over the list of cities, a function that lists the city, followed by the firsts of any pairs that have a second equal to the city.

```
(define (cluster L)
  (let* ((cities-with-duplicates  (sort (map second L) symbol<? ))
         (cities (remove-duplicates () cities-with-duplicates)))
```

```
      (map (lambda(city)
             (cons city
                   (map first (filter (lambda(pair) (equal? city (second pair))) L))))
           cities)))

(define (symbol<? x y) (string<? (symbol->string x) (symbol->string y)))

(define (remove-duplicates Previous L)
 (if (null? L)
    ()
    (if (equal? Previous (first L))
       (remove-duplicates Previous (rest L))
       (cons (first L) (remove-duplicates (first L) (rest L))))))
```

---

3. [5] Envision a surface on which there are placed several cards with only one side of each card visible. Each card is known to have a letter on one side and a digit on the other. For example, you might see:

A 1 8 G 4 B 9 E

Suppose that someone asserts:

"A vowel on the card implies that the number on the card is even."

Which cards must one flip over in order to establish whether or not the assertion is true? Answer both for the special case shown, and in general.

[5 points]: For the special case, A 1 9 and E would need to be flipped.

In general, every vowel showing needs to be flipped, to ascertain whether there is an even number on the other side, and every odd number needs to be flipped to ascertain whether there is a vowel on the other side (which would violate the assertion).

---

4. [5] Consider an n x n grid of bits used to represent images. A set of images can be classified by a single function that maps each such grid into {0, 1}. For a given n, how many different such functions are there?

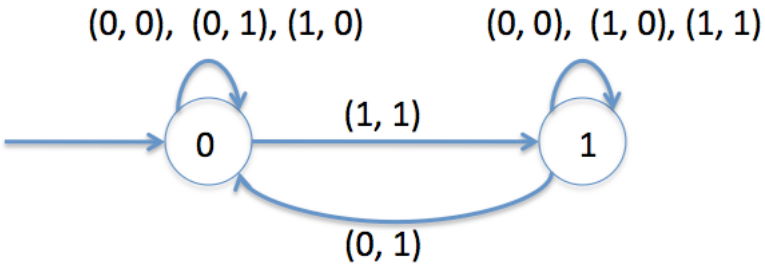
[5 points]: We show the power 2^k as 2^k. There are 2^(2^k) functions of k bits into {0, 1}. Here k is n^2. So there are 2^(2^(n^2)) functions in question.

5. [10] Give a logic implementation of a 1-bit register having two inputs:

- D (Data): A value to be stored in the register, provided L = 1.
- L (Load): If 1, the current value of D is stored in the register. If 0, the register maintains its current value.

Begin by showing the state diagram for the corresponding Moore machine. Then give the next-state function as a Karnaugh map. Finally, present a logic equation for the next state. Use S to represent the value of the state.

[5 points]: The next state as a function of the current state (left) and input (top) is:



		(D, L)			
		(0, 0)	(0, 1)	(1, 1)	(1, 0)
S	0	0	0	1	0
	1	1	0	1	1

[5 points]:

The table is arranged as a Karnaugh map, thus

next state = DL + SL'

Another possible answer is DL + SL' + SD, although the last term is redundant.

---

6. [5] Suppose we augment the previous register by one more input:

- C (Complement): Complement the current stored value (0 becomes 1, 1 becomes 0).

We stipulate that C and L will never be 1 at the same time, for consistency.

Give a next-state logic equation for this extension, taking advantage of the stipulation to simplify the logic.

[5 points]:

		(D, L)			
		(0, 0)	(0, 1)	(1, 1)	(1, 0)
(S, C)	(0, 0)	0	0	1	0
	(0, 1)	1	d	d	1
	(1, 1)	0	d	d	0
	(1, 0)	1	0	1	1

d = don't care (due to stipulation)

next state = DL + S'C + S C ' L'

---

7. [25] Consider the function re2dfa that converts a regular expression to a DFA accepting the strings represented by the regular expression, as discussed in conjunction with assignment 6. Suppose that we wish to extend the definition of re2dfa so that it recognizes **extended** regular expressions, defined to include two new operators, - and &:

(- R) means the **complement** of R, i.e. those strings, over the same alphabet, that are ***not*** in the language represented by R.

(& R S) means the **intersection** of R and S, those strings that are in ***both*** of the languages represented by R and by S.

Using the DFA representation from the homework, sketch scheme code for the handling of these new operators. Assume that DFA's for R and for S are constructible recursively, and that you have available code for all of the other regular expression operators. (Hint: Part of the strategy could involve the treatment of an expression as if another equivalent expression.)

[25 points]: Here is one solution:

Define a regular expression to be **clean** provided that it has no occurrences of – and &. This property can be tested recursively:

```
(define (clean? xre)
 (cond
   ((isEmpty  xre) #t)
   ((isLambda xre) #t)
   ((isLetter xre) #t)
   ((isComplement xre) #f)
   ((isIntersection xre) #f)
   ((isUnion  xre) (foldr (lambda (x y) (and x y))  #t (map clean? (rest xre))))
   ((isConcat xre) (foldr (lambda (x y) (and x y))  #t (map clean? (rest xre))))
   ((isStar   xre) (clean? (second xre)))
   (else ((x(error "invalid regular expression " xre))))))
```

For a clean regular expression, we can convert to a DFA using the derivatives method given in class. This forms the basis of a recursive conversion.

For a regular expression N that is **not clean**, we can **cleanse** it as follows:

If N is (- R), where R is already clean, then create a DFA from R. Change all of the accepting states to non-accepting and vice-versa, using complementDFA below. Then convert it back to a clean regular expression using the method of homework 6.

```
(define (complementDFA dfa)
  (map (lambda (row)
         (cons (first row) (cons (myNot (second row)) (rest (rest row)))))
       dfa))

(define (myNot x)
  (if (equal? 0 x) 1 0))
```

If N is (& R S), where R and S are clean, convert N syntactically to (- (+ (- R) (-S))) using DeMorgan's rule. Convert (-R) and (-S) to clean DFA's R' and

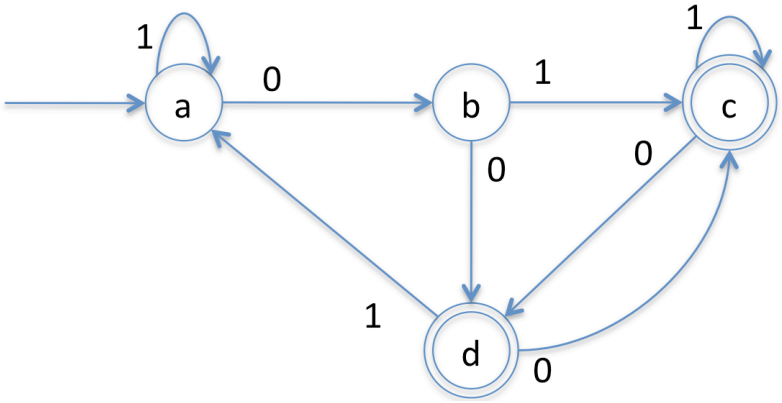
S'. So (+ R' S') is clean also. Then (- (+ R' S')) can be converted as above. (An alternative is to construct a product machine, but this is messier.)

The cleansing function can be implemented thus:

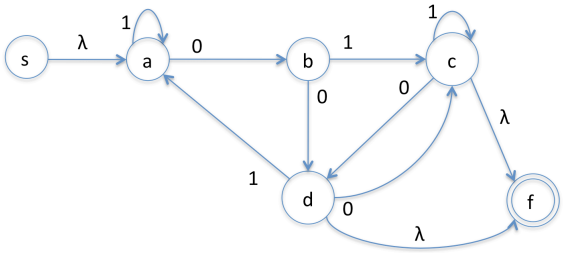
```
(define (cleanse alphabet xre)
 (cond
  ((clean? xre)        xre)
  ((isComplement xre)
       (dfa2re (complementDFA (re2dfa alphabet
               (cleanse alphabet (second xre))))))
  ((isIntersection xre)
       (dfa2re (complementDFA
               (re2dfa alphabet (
                       simplifyUnion
                       (map (lambda(x)
                        (cleanse alphabet (list complement-symbol x))) (rest xre)))))))
  ((isUnion xre)
        (cons union-symbol
               (map (lambda(x) (cleanse alphabet x)) (rest xre))))
  ((isConcat xre)
           (cons concat-symbol
               (map (lambda(x) (cleanse alphabet x)) (rest xre))))
  ((isStar xre)
               (list star-symbol (cleanse alphabet (second xre))))
  (else ((error "invalid extended regular expression " xre)))))
```

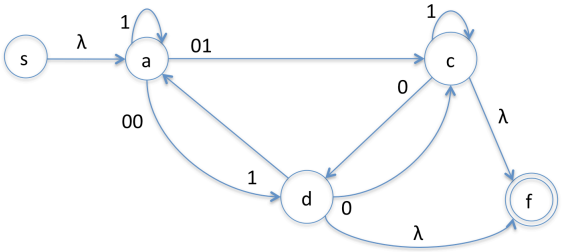
---

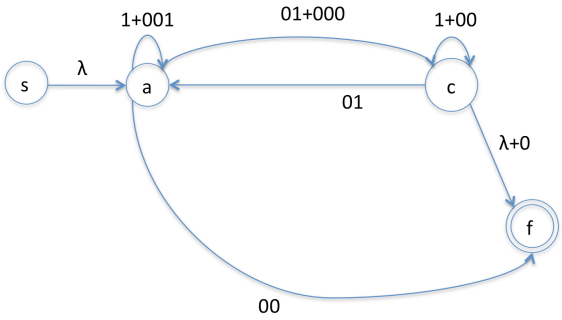
8. [10] Derive a regular expression for the following DFA, where **c** and **d** are the accepting states. Show your steps.



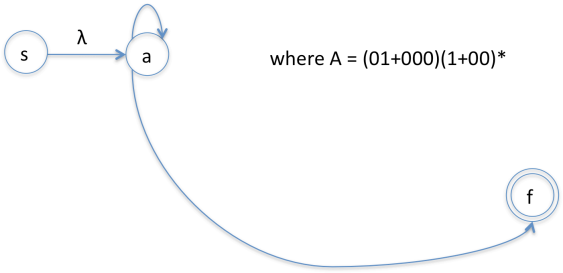
[10 points]: Add s and f, then eliminate nodes in order b, d, c, a:





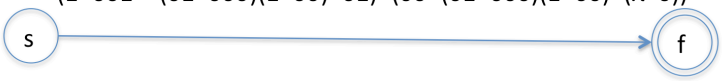


1+001 + A 01



where A = (01+000)(1+00)*

(1+001 + (01+000)(1+00)*01)*(00+(01+000)(1+00)*(λ+0))



The structure is (1 + 001 + A 01)*(00 + A (λ+0))

where A = (01+000)(1+00)*