Part I. (42 pts) Short Answer.

1. (2 pts) A function that takes a function as a parameter or yields a function as a result is known as a(n) higher-order function or functional form.

2. (2 pts) Efficiency is the primary concern of the imperative programming languages that are based directly on the underlying von Neumann architecture on which the program is executed.

3. (2 pts) An example of a functional programming language is Scheme or Lisp.

4. (2 pts) Which of the following is the best example of a fundamental statement in an imperative language that demonstrates its basis on the underlying von Neumann architecture?

   (a) assignment (it stores value into named memory locations)
   (b) function call
   (c) arithmetic operation
   (d) selection statement (e.g., if)

5. (4 pts) Mathematical functions produce no side effects. What does this mean? How does this relate to the concept of referential transparency? Mathematically, a function is just a mapping from domain elements to range elements. Applying the function on a domain element returns the corresponding the range element. In programming terms, calling the function with the same parameter (domain element) always produces the same result (range element). This is the definition of referential transparency. In imperative programming, functions can achieve side effects by additionally storing values to memory locations.

6. (4 pts) What are the two ways to interpret the list (A B C)? It could be a three element list of data or it could be a function call to A passing B and C as parameters.
7. (4 pts) Give a brief example of function composition. \( f(x) = 3 \times x \)

\[ g(x) = x + 2 \]
\[ h(x) = f(g(x)) \text{ is the composition of } f \text{ and } g \]
\[ h(x) = (3 \times x) + 2 \]

8. (6 pts) The basic process of computation is fundamentally different in a functional programming language than in an imperative language. Describe how it is different. In an imperative language, operations are done and the results are stored in variables for later use. Management of variables is a constant concern and source of complexity for imperative programming. In an FPL, variables are not necessary. Results are computed when needed by function calls.

9. (16 pts) What is the output of each of the following Scheme function calls?

(a) (2 pts) `(cons '((a)) '(b d))`
\(((a)) b d)\)

(b) (2 pts) `(append '((a)) '(b d))`
\((a) b d)\)

(c) (2 pts) `(apply cdr '(((a) b c) ((a b) c) (d e))))`
\(((a b) c) (d e))\)

(d) (2 pts) `((lambda(x y) (+ (- x y) (* x y))) 3 4)`
\(11\)

(e) (2 pts) `'+ (3 10)`
\((+ 3 10)\)
(f) (2 pts) (list? (list 'a '(b c) 'd))
    #t

10. (10 pts) Write a Scheme function called \textit{addList} that takes as input a simple list and returns the sum of the elements in the list if the list contains only numbers. The function must be recursive. (Recall, you do not have to worry about efficiency; that is, you can “recompute” a call to \textit{addList}.) Below are sample calls and expected output:

- \texttt{(addList '(3 2 3)) ----> 8}
- \texttt{(addList '(3 a 2)) ----> empty (a isn’t a number)}
- \texttt{(addList '()) ----> 0}

\begin{verbatim}
(define (addList lis)
  (cond
    ((list? lis) (cond
      ((null? lis) 0)
      (else (cond
        ((and (number? (car lis)) (addList (cdr lis))) (+ (car lis) (addList (cdr lis))))
        (else #f))))
    ((number? lis) lis)
    (else #f)))
\end{verbatim}
11. Write a Scheme function called *chop* that takes as input a list and returns all but the last element in the list. Below are sample calls and expected output:

- `(chop '(a b c d e)) ----> (a b c d)`
- `(chop '(a b (a b) e)) ----> (a b (a b))`

(a) (5 pts) You can use the built-in *reverse* function.

```scheme
(define (chop lis)
  (reverse (cdr (reverse lis))))
```

(b) (7 pts) You can NOT use the built-in *reverse* function.

```scheme
(define (chop lis)
  (cond
   ((not (list? lis)) '())
   ((null? lis) '())
   ((null? (cdr lis)) '())
   (else (cons (car lis) (chop (cdr lis))))))
```
12. (12 pts) Write a Scheme function called `flatten` that takes as input a list and returns a list that just contains the atoms of the original list. (That is, it removes all embedded parenthesis - nested sublists.) Below are sample calls and expected output:

```
(flatten '()) ----> empty
(flatten '2) ----> empty  (isn't a list)
(flatten '(a 3 b)) ----> (a 3 b)
(flatten '(((a) b) (c (d (e ((f)) g h))) i)) -->(a b c d e f g h i)
```

```
(define (flatten x)
  (cond
    ((not (list? x)) (list x))
    ((null? (cdr x)) (flatten (car x)))
    (else (append (flatten (car x)) (flatten (cdr x)))))))

(define (flatten2 x)
  (cond
    ((null? x) x)
    ((list? (car x)) (append (flatten2 (car x)) (flatten2 (cdr x))))
    (else (cons (car x) (flatten2 (cdr x))))))
```
13. (12 pts) Write a Scheme function called *myOr* that takes as input a simple list and returns true (#t) if any element in that list is true (#t). Otherwise *myOr* returns false. (You can NOT use the built-in *or* function.) Below are sample calls and expected output:

(\text{myOr} '('#f #f #f)) \quad \text{----> empty}

(\text{myOr} '('#f #t #f)) \quad \text{----> #t}

(\text{myOr} '()) \quad \text{----> empty}

\begin{verbatim}
(define (myOr lis)
  (cond
    ((not (list? lis)) #f)
    ((null? lis) #f)
    ((car lis) #t)
    (else (myOr (cdr lis))))
\end{verbatim}
14. Consider the following Scheme function:

```
(define (mystery l)
  (cond
    ((not (list? l)) empty)
    ((list? (car l)) (cons (car l) (mystery (cdr l))))
    (else (mystery (cdr l))))
```

(a) (2 pts) What would be returned when calling the function like this:

```
(mystery '())
```

Answer: `empty`

(b) (2 pts) What would be returned when calling the function like this:

```
(mystery 'a)
```

Answer: `empty`

(c) (6 pts) Show a “trace” of calls to mystery and indicate what would be returned when calling the function like this:

```
(mystery '((a) b (c d)))
```

Answer: `((a) (c d))` is returned.

A trace of calls to mystery (and returns):

```
(mystery '((a) b (c d)))
   l is a list, is not null, car of l is a list: (cons '(a) (mystery '(b (c d))))
   l is a list, is not null, car of l is atom: (mystery '(()
   l is a list, is not null, car of l is list: (cons '(c d) (mystery '())
   l is a list, is null: '()
   (cons 'c d '()) ---> ((c d))
   ((c d))
   (cons 'a '(() ---> ((a) (c d))
   ((a) (c d))
```

(d) (2 pts) Explain what the mystery function does. The function eliminates atoms from the list, leaving any sublists.

Answer: (It’s an atom bomb?!)
### Scheme Cheat Sheet

**Computer Science 3490 Programming Languages**  
**Test 2**  
**Scheme Cheat Sheet**  
**Tuesday, April 25, 2006**

| **cons** | constructs a list  
|----------|------------------  
| two parameters: first is atom or list second is list  
| (cons 'a (b c)) → (a b c)  
| (cons '(a) (b c)) → ((a) b c)  
| (cons '() (b c)) → (empty b c)  
| (cons '(a b) '()) → ((a b)) |

| **list** | constructs a list  
|----------|------------------  
| one or more parameters: atoms or lists  
| (list 'a 'b 'c) → (a b c)  
| (list '(a) 'b 'c) → ((a) b c)  
| (list 'a (b c)) → (a (b c))  
| (list '()) → empty  
| (list '() 'a) → (empty a) |

| **append** | constructs a list  
|-------------|------------------  
| two parameters: both are lists  
| (append '(a) (b c)) → (a b c)  
| (append '(a b) (c)) → (a b c)  
| (append '(b) (c)) → (b c)  
| (append '(a b) '()) → (a b)  
| (append '() (b c)) → (b c)  
| (append 'a '(b c)) → (a b c)  
| (append 'a '(b c)) → ILLEGAL  
| (append 'a '()) → ILLEGAL |

| **car** | first element of a list  
|----------|------------------  
| one parameter: a non-empty list  
| (car '(a b c)) → a  
| (car '(a b c)) → (a)  
| (car '()) → empty  
| (car 'a) → ILLEGAL |

| **cdr** | tail of a list  
|----------|------------------  
| one parameter: a non-empty list  
| (cdr '(a b c)) → (b c)  
| (cdr '(a b c)) → (c)  
| (cdr '()) → empty  
| (cdr 'a) → ILLEGAL |

| **apply** | calls a procedure on a list  
|-------------|------------------  
| two parameters: a procedure and a list  
| (apply + '(1 2 3)) → (+ 1 2 3) → 6  
| (apply car '(1 2 3)) → (car 1 2 3) → ILLEGAL  
| (apply car '(1 2 3)) → (car '(1 2 3)) → 1 |

| **eval** | evaluates a list as a function  
|----------|------------------  
| one parameter: a list  
| (eval '+ (2 3)) → (+ 2 3) → 5  
| (eval '(car '(1 2 3))) → (car '(1 2 3)) → 1 |

| **null?** | tests for empty list  
|-----------|------------------  
| one parameter: a list  
| (null? 'a) → ILLEGAL  
| (null? '()) → #t  
| (null? '(a)) → #f |

| **list?** | tests for a list  
|-----------|------------------  
| one parameter: atom or list  
| (list? 'a) → #f  
| (list? '(a)) → #t  
| (list '()) → #t |

| **number?** | tests for numeric atom  
|-------------|------------------  
| one parameter: atom or list  
| (number? 'a) → #f  
| (number? '2) → #t  
| (number? 'a) → #f  
| (number? '()) → #f |

| **eq?** | tests two atoms for equality  
|---------|------------------  
| two parameters: atoms  
| (eq? 'a 'b) → #f  
| (eq? 'b 'b) → #t  
| (eq? 'b '(b)) → ILLEGAL  
| (eq? '3 '3) → #t  
| (eq? '4 '5) → #f |

| **<** | tests numeric atoms  
|-------|------------------  
| > | tests numeric atoms  
| <= | tests numeric atoms  
| >= | tests numeric atoms  
| = | tests numeric atoms  
| <> | tests numeric atoms |

| **and** | tests atoms for truthness  
|--------|------------------  
| two parameters: atoms (anything not equal to #f is #t)  
| (and 'a 'b) → #f  
| (and 'b 'b) → #t  
| (and 'b '(b)) → ILLEGAL  
| (and '3 '3) → #t  
| (and '4 '5) → #f |

| **or** | tests atoms for truthness  
|--------|------------------  
| two parameters: atoms (anything not equal to #f is #t)  
| (or 'a 'b) → #f  
| (or 'b 'b) → #f  
| (or 'b '(b)) → ILLEGAL  
| (or '3 '3) → #t  
| (or '4 '5) → #f |

| **not** | negates atom truthness  
|--------|------------------  
| one parameter: atom (anything not equal to #f is #t)  
| (not 'a) → #f  
| (not 't) → #f  
| (not '#f) → #t  
| (not 'a) → #f  
| (not 't) → #f  
| (not '#f) → #t |