1	25/
2	20/
3	30/
4	40/
5	15/
6	15/
7	15/
8	15/
9	15/
200/	

CMSC 331 Midterm Exam Fall 2008

Name: _____

UMBC username:_____

You will have seventy-five (75) minutes to complete this closed book/notes exam. Use the backs of these pages if you need more room for your answers. Describe any assumptions you make in solving a problem. We reserve the right to assign partial credit, and to deduct points for answers that are needlessly wordy.

1. True/False (25 pts: 2/2/2/2/2/2/2/2/2/2/2/2/2 + 1 freebie)

For each of the following questions, circle T (true) or F (false).

T F FORTRAN was one of the first programming languages in the functional programming paradigm. FALSE

T F Prolog was developed primarily for applications in science and engineering. FALSE

T F An advantage of using a compiler is that it generally produces portable code able to fun on many different hardware and software platforms. FALSE

T F Compilers are generally more difficult to implement than interpreters. TRUE

T F Lisp was originally developed for Artificial Intelligence problems. TRUE

T F In a BNF grammar, a terminal symbol can not appear on the left hand side of a rule. TRUE

T F The EBNF (extended BNF) notation allows one to describe languages that can not be specified in BNF. FALSE

T F A grammar with a finite set of terminal symbols can only define a finite language. FALSE

T F A grammar G is ambiguous if there are two different derivations for at least one sentence in the language. FALSE

T F Associativity rules only apply to operators of the same precedence level. TRUE

T F A grammar with left recursive rules can not be directly used to implement a recursive descent parser. TRUE

T F The Unix Lex program is used to define scanners that produce a stream of tokens from a stream of characters. TRUE

2. General multiple-choice questions (20 pts: 4/4/4/4/4)

1.7 Which of the following Scheme expressions would be interpreted as false when evaluated: (a) NIL; (b) -1; (c) (CAR '(0 1)); (d) '(); (e) none of the above. **(E)**

1.8 Which phrase best describes the variable scoping used in Scheme? (a) wide scoping; (b) narrow scoping; (c) lexical scoping; (d) dynamic scoping; (e) structured scoping; (f) denotational scoping; (g) none of the above. **(C)**

1.9 In Scheme a lambda expression represents a (a) abstract class; (b) variable type; (c) function; (d) conditional; (e) cons cell. **(C)**

1.10 Scheme macros are primarily used to define: (a) dynamically scoped environments; (b) closures; (c) functions that don't evaluate all of their arguments; (d) reflective programs. **(C)**

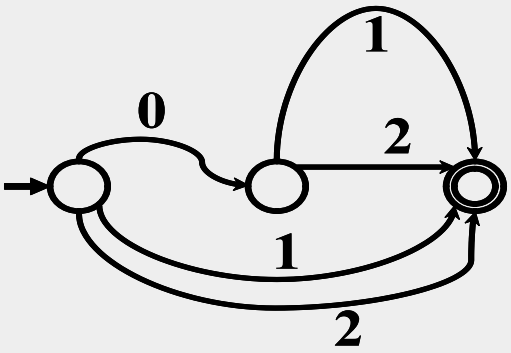
1.11 Which of the following is not considered a functional programming language? (a) ML; (b) Haskell; (c) Smalltalk; (d) Scheme; (e) Lisp. **(C)**

3. Regular expressions (40 pts: 20/20)

This problem asks you to define a simple language using a deterministic finite automaton (DFA), a regular expression (RE) and a context free grammar (CFG). Use the normal conventions for DFAs, REs and CFGs shown in the example to the right.

This problem asks you to define a simple language for numbers that includes both integers and numbers with a decimal part that follows these rules:

- leading zeros are not allowed except for the string "0"
- trailing zeros in the decimal part are not allowed
- a number should not end in a decimal point



This example shows a DFA for the regular expression `0?[12]` and is also be described by the CFG:

```
S -> 0A | A
A -> 1 | 2
```

Here are examples of strings that should be in and out of the language:

IN: 0, 130, .203, 12.32, .01, .1
OUT: 01, 01.2, 01., 0., 0.23, 12.0, 1.20, 12.

(a) Draw a DFA for this language. Feel free to define a class of characters using a notation like the following, which represents any single digit between 0 and 9 and to put such a class name on an arc in your DFA.

```
DIG: [0-9]
```

(b) Write a regular expression using a notation like the following, which describes a simple RE for email addresses. Recall that * indicates any number of repetitions and a + indicates one or more repetitions. Use parentheses to group things. A vertical bar separates alternatives.

```
LET: [a-zA-Z]
DIG: [0-9]
LET ( LET | DIG )* @ ( LET | DIG )+ ( \. ( LET | DIG )+ )+
```

4. Operators (40 pts: 5/5/5/5/5/15)

Consider the following BNF grammar for a language with two infix operators represented by & and @. Note: the order of the rules in a grammar is not significant.

<blooper> ::= a
<blooper> ::= b
<blooper> ::= c
<super> ::= <blooper> & <super>
<super> ::= <blooper>
<duper> ::= <duper> @ <super>
<duper> ::= <super>

A. What are the (i) terminal symbols and (ii) non-terminal symbols in this grammar.
Terminal = {a,b,c} non-terminal={super, blooper, duper}

B. What is the associativity of the & operator: (a) left; (b) right; (c) neither. **(B)**
C. What is the associativity of the @ operator: (a) left; (b) right; (c) neither. **(A)**
D. Which operator has higher precedence: (a) &; (b) @; (c) neither. **(A)**
E. The grammar is (a) left recursive; (b) right recursive; (c) both left and right recursive; (d) neither left nor right recursive. **(C)**
F. Draw a parse tree for the following string: a & b & c @ a @ b

5. Constructing s-expressions (15: 5/5/5)

Consider the Scheme data Structure that when printed looks like (a (b) c)

5.1 Give a Scheme expression using only the **cons** function that will create this list. Use the variable **empty** to represent the empty list.

(cons 'A (cons (cons 'B empty) (cons 'C empty)))

5.2 Give a Scheme expression using only the LIST function that will create this list.

(list 'A (list 'B) 'C)

5.3 Assuming that we've done **(define X '(a (b) c))** give Scheme expression using only the functions CAR and CDR and the variable X that would return each of the three symbols in the list.

symbol	*s-expression to return the symbol*
A	**(car X)**
B	**(car (car (cdr X)))**
C	**(car (cdr (cdr X)))**

6. Writing a Scheme function (15: 5/5/5)

This question asks you to write two versions of the Scheme function SUMLIST that takes a list of numbers and returns their sum, as shown in the box to the right. Assume that your function is always passed a valid argument.

```
> (sumlist empty)
0
> (sumlist '(100))
100
> (sumlist '(1 2 3 4 5))
15
>
(define (reduce f v l)
  (if (null? l)
      v
      (f (first l)
         (reduce f v (rest l))))
>
```

(a) Write **sumlist** as a simple recursive function.

Comment: here are two variations:

(define (sumlist l) (if (null l) 0 (+ (car l) (sumlist (cdr l)))))

(define (sumlist l) (cond ((null? l) 0) (#t (+ (car l) (sumlist (cdr l))))))

(b) Write **sumlist** without using recursion or iteration, but using either APPLY.

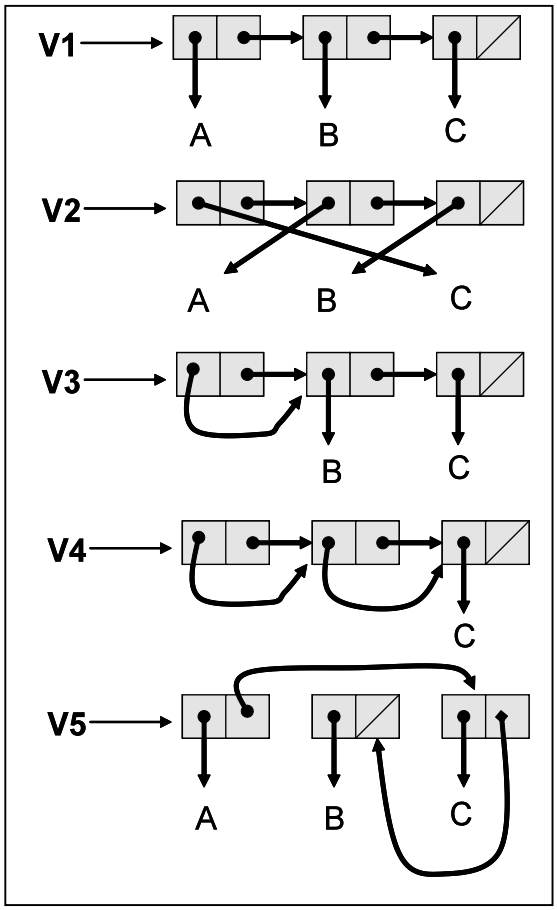
(define (sumlist l) (apply + l))

(c) Write sumlist using + and the reduce function discussed in class and define in the box to the right.

7. List structures (15: 10/5)

(7a) Show what would be printed for each of the five list structures to the right.

	What would be printed?
V1	**(A B C)**
V2	**(C A B)**
V3	**((B C) B C)**
V4	**(((C) C) (C) C)**
V5	**(A C B)**



(7b) Draw a box and pointer diagram for V6: (((A)))

```
> (define (filter F L)
    ; returns elements of  L for which F is true
    (cond ((null? L) empty)
          ((F (first L))
           (cons (first L)  (filter F (rest L))))
          (#t  (filter F (rest L)))))
filter
> (filter even? '(1 2 3 4 5 6))
(2 4 6)
```

8. Functional programming I (15: 5/5/5)

Recall the filter function shown to the right. It takes two arguments: F, a single argument predicate function and a list L and returns a list of L's elements for which F is true. Suppose the variable **peeps** is bound to a list of sub-lists representing a person's name, age, sex and major, as in:

```
(define peeps '( (john 19 male cmsc)  (mary 23
female cmpe)  (sue 20 female ifsm)
                (bill 21 male cmsc)  (alice 18 female cmsc)  …  )
```

Using filter and anonymous functions defined as lambda expressions, give expressions that return a list of people in **peeps** that:

Comment: we need to filter the list with a function that takes a person structure (e.g., (john 19 male cmsc) and returns true it it matches the desired constraints and false otherwise. It's just a matter of writing a boolean condition on parts of the person structure. There are several ways to write each of the expressions below.

(a) Are aged 21 or older.

```
(filter (lambda (p) (> (car (cdr p)) 20) peeps)
```

(b) Are women aged 21 or older

```
(filter (lambda (p) (and  (> (car (cdr p)) 20) (equal? (caddr p) 'female) )) peeps)
```

(c) are women CMPE or CMSC majors older than 18

```
(filter (lambda (p) (and  (> (car (cdr p)) 18)
                          (equal? (caddr p) 'female)
                          (member (cadddr p) '(cmsc cmpe))))
        peeps)
```

9. Functional programming II (15: 3/3/4/5)

Another aspect of functional programming is the ability to write functions to create new functions. Consider the function XYZZY defined as:

```
(define (XYZZY F)  (LAMBDA (X) (F (F X))))
```

(a) With what type of arguments should xyzzy be called?
A function of one argument

(b) What type of thing does xyzzy return?
A function of one argument

© Describe in a sentence what xyzzy does?
Given a unary function F, XYZZY will return a new function which is equal to F composed with itself.

(d) What will this expression return: (map (XYZZY (lambda (X) (* X X))) '(1 2 3 4))
Comment: (lambda (x) (* x x)) *is just an anonymous function that squares its numerical argument. So XYZZY applied to this returns a function that takes the square of the square of its argument, i.e., it raises its argument to the fourth power. So, mapping this down the list of the first six integers produces the following answer.* **(1 16 81 256)**