1	40/
2	30/
3	45/
4	30/
5	30/
6	30/
7	20/

225/	

CMSC 331 Midterm Exam, Fall 2010 a

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 [40]

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

T F **1.1** COBOL was designed as a programming language for scientific and engineering applications. **FALSE**

T F **1.2** The *procedural* programming paradigm treats procedures as first class objects. **FALSE**

T F **1.3** The "Von Neumann" computer architecture is still used as the basis for most computers today. **TRUE**

T F **1.4** One of the advantages of interpreted over compiled languages is that they tend to offer more run time debugging support. **TRUE**

T F **1.5** Any finite language can be defined by a regular expression. **TRUE**

T F **1.6** Attribute grammars can specify languages that can not be specified using a context free grammar. **TRUE**

T F **1.7** A recursive descent parser can not directly use a grammar that has right recursive rules. **FALSE**

T F **1.8** The lexical structure of complex programming languages like Java can not be defined using regular expressions. **FALSE**

T F **1.9** A non-deterministic finite automaton for a regular language is generally easier to write than a deterministic one, but harder to apply to a string to see if it matches. **TRUE**

T F **1.10** If the grammar for a language is unambiguous, then there is only one way to parse each valid sentence in that language. **TRUE**

T F **1.11** A BNF grammar can not contain both left-recursive and right-recursive rules. **FALSE**

T F **1.12** The EBNF notation allows one to define grammars that can not be defined using the simpler BNF notation. **FALSE**

T F **1.13** The order of production rules in a grammar is not significant, i.e., two grammars with identical rules but given in different order will always define the same language. **TRUE**

T F **1.14** An operator's precedence determines whether it associates to the left or right. **FALSE**

T F **1.15** Specifying how *else clauses* match with the right *if* keyword is done by adjusting the precedence of the i*f*, *then* and *else* operators. **FALSE**

T F **1.16** Scheme's simple grammar eliminates the need to define operator precedence. **TRUE**

T F **1.17** The idea behind axiomatic semantics is to define the meaning of statements in a programming language by translating them into statements in another language. **FALSE**

T F **1.18** In Scheme, evaluating a symbol requires looking up the value assigned to it as a variable. **TRUE**

T F **1.19** In Scheme, a predicate always returns either #t or #f. **FALSE**

T F **1.20** Scheme uses dynamic scoping to resolve the value of a free (i.e., non-local) variable. **FALSE**

2. General multiple-choice questions [30]

Circle all of the correct answers and only the correct answers.

2.1 Which of the following is considered an object-oriented programming language? (a) ML; (b) Haskell; (c) Smalltalk; (d) Scheme; (e) C# (f) Java (g) Algol **(C, E, F)**

2.2 *Left factoring* is a technique that can be used to (a) prepare a grammar for use in a recursive descent parser; (b) produce a left most derivation of a string from a grammar; (c) remove left recursion from a grammar; (d) factor out left associative operators; (e) eliminate a non-terminal from the left side of a grammar rule; (f) all of the previous answers; (g) none of the previous answers. **(A,C)**

2.3 A LL(1) parser (a) processes the input symbols from left to right; (b) produces a left-most derivation; (c) looks ahead at most one input symbol before knowing what action to take; (d) takes time proportional to the cube of the number of input symbols **(A, B, C)**

2.4 Attribute grammars are used to (a) model the basic syntax of a programming language; (b) specify non-finite state machines; (c) specify the static semantics of a programming language; (d) specify the dynamic semantics of a programming language; (e) create parsing tables for LR(k) parsers. **(C)**

2.5 Which of the following parsing algorithms use a bottom up approach as opposed to a top-down one: (a) recursive descent; (b) LL(k); (c) LR(k). **(C)**

2.6 In Scheme, a tail-recursive algorithm is generally better than a non-tail recursive algorithm because (a) it can be run without growing the stack; (b) it is easier to understand; (c) it has no side-effects; (d) all of the above. **(A)**

2.7 Tail-call optimization (a) is done in all programming languages; (b) turns recursion into iteration; (c) can speed up program execution; (d) can introduce exceptions; (d) can prevent stack overflow. **(B, C, D)**

2.8 In Scheme, a free variable in a function is looked up in (a) the global environment; (b) the environment in which the function was defined; (c) the environment from which the function was called; (d) all active environments. **(B)**

2.9 Which of the following Scheme expressions would be interpreted as false when evaluated: (a) 0; (b) -1; (c) null; (d) #f; (e) (lambda () #f); (f) (not -1); (g) ((lambda () #f)) **(D, F, G)**

2.10 In Scheme, evaluating a lambda expression always returns an (a) environment; (b) variable type; (c) function; (d) conditional; (e) pair. **(C)**

3. Operators [45]

Given the following BNF grammar for a language with two infix operators represented by # and $.

```
<bar> ::= <baz>
<foo> ::= <bar> $ <foo>
<baz> ::= ( <foo> )
<bar> ::= <bar> # <baz>
<baz> ::=  x | y
<foo> ::= <bar>
```

a) [5] Which operator has higher precedence: (i) $; (ii) #; (iii) neither; (iv) both **(ii)**

b) [5] What is the associativity of the $ operator: (i) left; (ii) right; (iii) neither **(ii)**

c) [5] What is the associativity of the # operator: (i) left; (ii) right; (iii) neither **(i)**

d) [5] Assuming that the start symbol is <foo>, does this grammar define a finite or infinite language? **infinite**

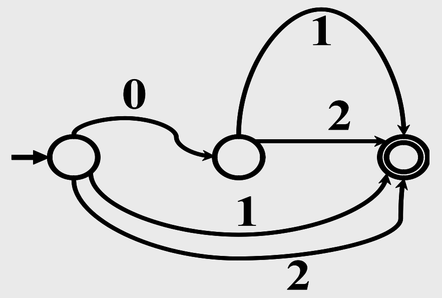
e) [5] Assuming that the start symbol is <foo>, is this grammar: (i) ambiguous or (ii) unambiguous? **(ii)**

f) [20] Give a parse tree for the following string:

```
x $ x # y # ( y $ x )
```

to be supplied

4. Regular expressions [30]



This example shows a DFA for the regular expression `0?[12]`

The UMBC registrar uses a code for courses consisting of three parts:

- A four letter upper-case program abbreviation (e.g., CMSC, CMPE, HIST)
- A three digit course number that can't begin with a zero or a nine (e.g., 331, 104)
- An optional upper or lower case letter (e.g., H, A, w)

Examples of legal codes are CMSC331H and CMSC491 and of illegal codes are CS331 and CMSC001.

(a) [15] Draw a deterministic finite automaton (DFA) for this language. Feel free to define a class of characters using a notation like the following, which represents a letter and a single digit and to put such a class name on an arc in your DFA.

```
LET: [a-zA-Z]
DIG: [0-9]
```

(c) [15] Write an equivalent regular expression for your DFA. Use a notation in which a '*' indicates any number of repetitions, '+' indicates one or more repetitions, '?' means zero or one repetitions, parentheses group things, a vertical bar separates alternatives, etc., as in the following example.

```
LET: [a-zA-Z]
((mr|mrs|ms|dr)\.\t+)? LET+ (\t+ LET+)*
```

UC: [A-Z]
LET: [a-zA-Z]
D1: [1-8]
D2: [0-9]

UC UC UC UC D1 D2 D2 LET?

DFA to be supplied

5. Constructing s-expressions [30]

Consider the Scheme data Structure that when printed looks like `((1 (2) 3))`

5.1 [5] Give a Scheme expression using only the **cons** function that will create this list. Use the variable **null** for the empty list.

```
(cons (cons 1 (cons (cons 2 null) (cons 3 null))) null)
```

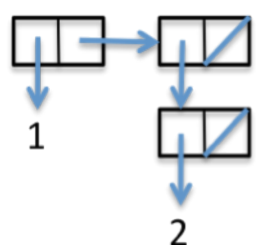
5.2 [5] Give a Scheme expression using only the **list** function that will create this list. Use **null** for the empty list.

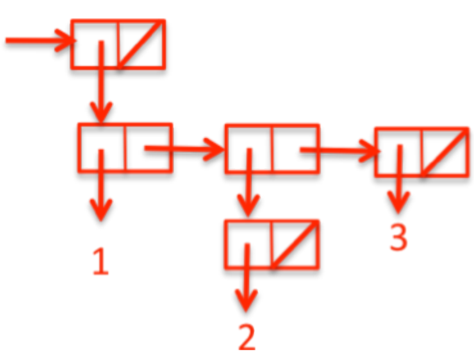
```
(list (list 1 (list 2) 3))
```

5.3 [10] Assuming that we've done **`(define x '((1 (2) 3)))`** give a Scheme expression using only the functions **`car`** and **`cdr`** and variable **`x`** that returns the three symbols in the list.

symbol	*s-expression to return the symbol*
1	(car (car x))
2	(car (car (cdr (car x))))
3	(car (cdr (cdr (car x))))

5.4 [10] Draw a "box and pointer" diagram showing how the list **`((1 (2) 3)))`** is represented in pairs. The figure to the right shows an example of the diagram format you should use. This example represents the list (1 (2)).





6. Scheme I [30]

Common Lisp has a built-in function mapcan. The Scheme counterpart could be defined as follows:

```
(define (mapcan f l)
  (if (null? l)
      null
      (append (f (car l))
              (mapcan f (cdr l)))))
```

(a) [10] What will (mapcan list '(1 2 3 4 5 6)) return?

```
(1 2 3 4 5 6)
```

(b) [10] What will (mapcan (lambda (x) (if (even? x) (list x) null)) '(1 2 3 4 5 6)) return?

```
(2 4 6)
```

(c) [10] Redefine mapcan in Scheme without using recursion by using the apply, append and map functions.

```
(define (mapcan f l) (apply append (map f l)))
```

7. Scheme II [20]

Consider a function *insert* with three arguments: an arbitrary s-expression, a proper list, and a positive integer. The function returns a new list that is the result of inserting the expression into the list at the position specified by the third argument. Note that positions begin with zero. For example,

```
> (insert 'X '(a b c) 3)
(a b c X d)
> (insert '(X) '(a b c) 1)
(a (X) b c)
> insert 'X '(a b c) 0)
(X a b c)
```

Here is an incomplete definition of the function. Give code expressions for <S1>, <S2> and <S3> that will complete it.

```
(define (insert expr lst pos)
    ;; Returns a list like proper list lst but with expr inserted at
    ;; the position given by positive integer pos. e.g.: (insert 'X
    ;; '(a b c) 2) => (a b X c)
    (cond (<S1> (cons expr lst))
          ((null? lst) <S2> )
          (else <S3>)))
```

<S1>	**(= pos 0) or (< pos 1) or (eq? pos 0) oe (equal? pos 0) or equivalent**
<S2>	**(cons expr null) or (list expr)**
<S3>	**(cons (car lst) (insert expr (cdr lst) (- pos 1))) or equivalent**