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

225/	

CMSC 331 Midterm Exam, Fall 2010 b

Name: ________________________________

UMBC username:_____________________________

You will have seventy-five (75) minutes to complete this closed book/notes exam. Use the backs of pages if you need more room for your answers. Describe any assumptions you make in solving a problem. We may assign partial credit and 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** FORTRAN was designed as a programming language for scientific and engineering applications. **TRUE**

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

T F **1.3** The "Von Neumann" computer architecture dominated the early days of computing but was largely replaced by new designs in the 1980s. **FALSE**

T F **1.4** An advantage of a compiler over an interpreter is that it generally produces portable code able to run on many different hardware and software platforms. **FALSE**

T F **1.5** A grammar with a finite number of non-terminal and terminal symbols and also a finite number of rules can specify an infinite language. **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't directly use a grammar that has left recursive rules. **TRUE**

T F **1.8** The lexical structure of complex programming languages like Python can be defined using regular expressions. **TRUE**

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

T F **1.10** If the grammar for a language is ambiguous, then there is more than one way to parse every valid sentence in that language. **FALSE**

T F **1.11** A BNF grammar can contain both left-recursive and right-recursive rules. **TRUE**

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 a grammar's production rules 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 *if*, *then* and *else* operators. **FALSE**

T F **1.16** Scheme's simple grammar makes it easy to define functions that take a variable number of arguments. **TRUE**

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

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** A Scheme predicate function is one whose returned value can be interpreted as a Boolean. **TRUE**

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) remove left recursion from a grammar; (b) factor out left associative operators; (c) eliminate a non-terminal from the left side of a grammar rule; (d) prepare a grammar for use in a recursive descent parser; (e) produce a left most derivation of a string from a grammar; (f) all of the previous answers; (g) none of the previous answers. **(A, D)**

2.3 A LR(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, C)**

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

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

2.6 Scheme optimizes "tail calls" to (a) execute recursion as iteration; (b) speed up program execution; (c) improve program readability; (d) lessen the chance of stack overflow, (e) all of the above. **(A, B, D)**

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

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 true when evaluated: (a) 0; (b) -1; (c) null; (d) #f; (e) (lambda () #t) (f) (not (not 1)) ; (g) (not -1). **(A, B, C, E, F)**

2.10 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)**

3. Operators [45]

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

```
<tic> ::= <toe>
<tac> ::= <tic> $ <tac>
<toe> ::= ( <tac> )
<tic> ::= <tic> # <toe>
<tac> ::= <tic>
<toe> ::=  x | y
```

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 `<tac>`, does this grammar define a finite or infinite language? **infinite**

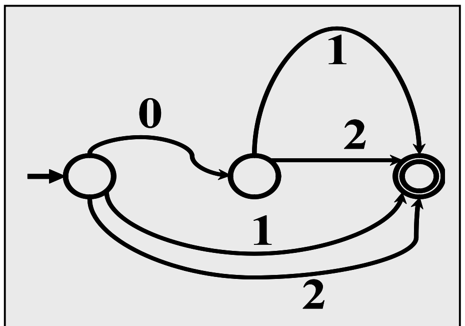
e) [5] Assuming that the start symbol is `<tac>`, is this grammar: (i) ambiguous or (ii) unambiguous? **(ii)**

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 between 100 and 899 (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?
```

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) null)) (cons (cons 3 null) null))
```

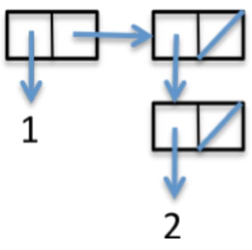
5.2 [5] Give a Scheme expression using only the **list** function that will create this list. Use **null** for the empty list.

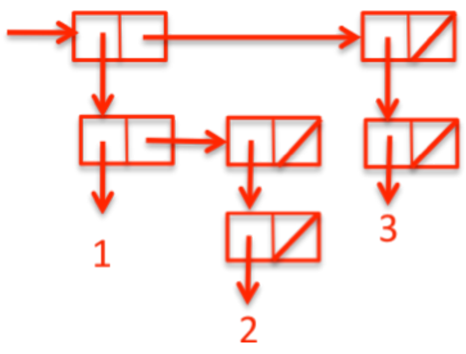
```
(list (list 1 (list 2)) (list 3))
```

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 (car (cdr 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 maplist. The Scheme counter part could be defined as follows:

```
(define (maplist f l)
  (if (null? l)
      null
      (append (f l)
              (maplist f (cdr l)))))
```

(a) [10] What will (maplist list '(1 2 3)) return?

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

(b) [10] What will (maplist (lambda (x) x) '(1 2 3)) return?

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

(c) *[10]* What will (maplist (lambda (x) (list (length x))) '(1 2 3))

(3 2 1)

7. Scheme II [20]

Consider a function prefix with two arguments, both of which are proper lists. It returns true if the first is a prefix of the second.

```
> (starts null '(1 2 3 4))
#t
> (starts '(1 2) '(1 2 3 4))
#t
> (starts '(1 2 3 4) '(1 2 3))
#f
> (starts '(1 2 x) '(1 2 3 4))
#f
> (starts '(1 2) '(1 2))
#t
> (starts ‘(1 2) '())
#f
```

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

```
(define (starts one two)
  (cond ((null? one) <S1> )
        ((null? two) <S2> )
        (<S3>
         <S4>)
        (else <S5> )))
```

<S1>	#t
<S2>	#f
<S3>	(equal? (car one) (car two))
<S4>	(starts (cdr one)(cdr two))
<S5>	#f