1	14/
2	20/
3	14/
4	12/
5	12/
6	16/
7	12/

	100/

CMSC 331 Midterm Exam, Spring 2011

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 [14]

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

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

T F **1.2** The *procedural* programming paradigm treats user-defined procedures as first class objects.

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

T F **1.4** Any finite language can be defined by a regular expression.

T F **1.5** Attribute grammars can specify languages that cannot be specified using a context free grammar alone.

T F **1.6** A recursive descent parser cannot be used to parse strings generated by an ambiguous grammar.

T F **1.7** The entire syntax of complex programming languages like Java cannot be defined using regular expressions.

T F **1.8** A non-deterministic finite automaton for a regular language generally has fewer states than a deterministic one, but is harder to apply to a string to see if it matches.

T F **1.9** If the grammar for a language is unambiguous, then there is only one way to parse each valid sentence in that language.

T F **1.10** The EBNF notation allows one to define grammars that cannot be defined using the simpler BNF notation.

T F **1.11** An operator's precedence determines whether it associates to the left or right.

T F **1.12** Specifying how *else clauses* match with the right *if* keyword is done by adjusting the precedence of the i*f*, *then* and *else* operators.

T F **1.13** The prefix operations of Lisp-like languages eliminates the need to define operator precedence.

T F **1.14** In Scheme, evaluating a macro requires looking up the pattern assigned to that macro.

2. General multiple-choice questions [20]

Place a check mark next to all of the correct answers and only the correct answers.

2.1 Which of the following is considered an object-oriented programming language?
____(a) Lisp;
____(b) C++;
____(c) Pascal;
____(d) Scheme;
____(e) Java
____(g) Algol

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 terminal from the left side of a grammar rule;
____(f) none of the previous answers.

2.3 In general, a recursive descent parser
____(a) processes the input symbols from left to right;
____(b) produces an abstract syntax tree;
____(c) looks ahead at most one input symbol before knowing what action to take;
____(d) takes more time than other parsers for the same language.

2.4 Attribute grammars are used to
____(a) model the basic syntax of a programming language;
____(b) specify non-deterministic finite state machines;
____(c) specify the static semantics of a programming language;
____(d) specify the dynamic semantics of a programming language.

2.5 The purpose of axiomatic semantics is to
____(a) prevent and detect logic errors in programs;
____(b) cause functional programs to use less memory;
____(c) make sure loop invariants hold during loops;
____(d) prove that programs are correct;
____(e) all of the above.

2.6 In functional programming languages, a tail-recursive algorithm is generally better than a non-tail recursive algorithm because
____(a) it can be run without growing the stack to excess;
____(b) it is easier to understand;
____(c) it is faster;
____(d) all of the above.

2.7 In a static-scoped language like Scheme, a free variable in a function is looked up in
____(a) the local environment(s), then the global environment;
____(b) the local environment(s), then in the environment(s) of the calling function;
____(c) the global environment, since free variables are global;
____(d) none of the above.

2.8 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) '()
____(g) ((lambda () #f)).

2.9 In Lisp (or Scheme) the cons operators is used to
____(a) create dotted pairs
____(b) add atoms at the end of a list
____(c) access the address register on older computers
____(d) create other data structures such as hash tables

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

3. Operators [14]

In the BNF grammar shown, x and y are terminals, all non-terminals are in angle brackets, and <tac> is the start symbol. This language has two infix operators represented by # and $.

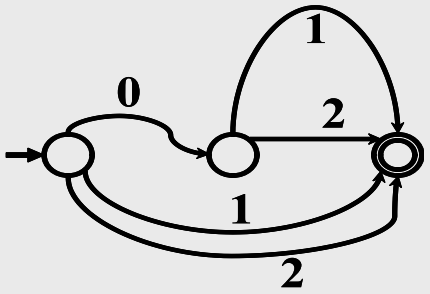
a) [2] Which operator has higher precedence?
____(i) $;
____(ii) #;
____(iii) we can't tell

b) [2] What is the associativity of the $ operator:
____(i) left;
____(ii) right;
____(iii) we can't tell

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

c) [10] Give a parse tree for the following string: `x # x $ y`

4. Regular expressions [10]



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

As you know, passwords are the first and most important step in computer security. To have a password of sufficient strength is critical. A password is just strong enough if it contains at least one lower-case letter, one upper-case letter, and one digit.

Examples of good passwords: X12ab, doraymeABC123helloMJ, 7SEVENseven

Examples of bad passwords: password, mycatSam, 1password

Consider the language consisting of just strong enough passwords, that is, strings that have AT LEAST one lower-case letter, one upper-case letter, and one digit. The alphabet is the lower-case letters a-z, the upper-case letters A-Z, and the digits 0-9. No other characters need be considered.

[10] Draw a deterministic finite automaton (DFA) for this language. Feel free to define a class of characters using a notation like the following, so that you can use such a class name on an arc in your DFA. (Hint: you might need to have several states.)

```
DIG: [0-9]
LOWER: [a-z]
UPPER: [A-Z]
```

5. Constructing s-expressions [12]

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

5.1 [6] Give a Scheme expression using only the **cons** function that will create this list. Use the variable **null** for the empty list.

5.2 [6] Assuming that we've done **`(define x '((3 (2)) (1))`** 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	
2	
3	

6. Lisp and Scheme I [16]

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 <S2>, <S3>, <S4> and <S5> that will complete it.

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

<S1>	**#t (as an example)**
<S2>	
<S3>	
<S4>	
<S5>	

7. Lisp and Scheme II [12]

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)
> (insert '(X) '(a b c) 1)
(a (X) b c)
> (insert 'X '(a b c) 0)
(X a b c)
> (insert `X `(a b c) 4)
#f
```

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>	
<S2>	
<S3>	