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 if, 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

c) [10] Give a parse tree for the following string:  \( x \ # \ x \ $ \ y \)
4. Regular expressions [10]

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.

<table>
<thead>
<tr>
<th>symbol</th>
<th>s-expression to return the symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
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 3 4) '(1 2 x))
#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> )))
```

<table>
<thead>
<tr>
<th>&lt;s1&gt;</th>
<th>#t</th>
<th>(as an example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;S2&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;S3&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;S4&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;S5&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Consider a function \textit{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,

\begin{verbatim}
> (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
\end{verbatim}

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

\begin{verbatim}
(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>)))
\end{verbatim}

\begin{tabular}{|c|}
\hline
\(<S1>\) \\
\hline
\(<S2>\) \\
\hline
\(<S3>\) \\
\hline
\end{tabular}