Please show all work to receive ANY credit!!!!

1. (30 points, 6 points each) Below are a list of seven concepts that are important for compilers. You are to choose 5 of the seven concepts and provide the following information:

   i) a definition of the concept 
   ii) a discussion of why the concept is important and how the concept is used in the design and implementation of a compiler 
   iii) an example of the concept 

I will only grade five, so there’s no advantage to answering more! Circle the letters in the list of a to g given below for the five concepts that you want me to grade.

   a. transition diagrams 
   b. panic mode error recovery 
   c. regular expressions 
   d. token, pattern, and lexeme 
   e. context free grammar 
   f. phase level error recovery 
   g. finite automaton
2. (15 points) Construct a DFA for the regular expression: ((cab)* | (bc)*)a

3. (20 points) Redesign the grammar given below into an equivalent form that is suitable for top down parsing:

\[
\begin{align*}
S & \rightarrow (L) \mid Eb \mid a \\
L & \rightarrow L , S \mid S \mid c \\
E & \rightarrow E \mid Z \mid Le \mid Ed
\end{align*}
\]

4. (15 points) Consider the grammar segment shown below from the first laboratory project:

\[
\begin{align*}
<\text{what-queries}> & : = \text{WHAT} <\text{are-is}> <\text{what-options}> \text{OF} <\text{shared-targets}> \\
<\text{are-is}> & : = \text{ARE IS} \\
<\text{what-options}> & : = \text{PARAMETERS} \mid \text{RETURN_TYPE} \mid \text{LOCAL_VARS} \mid \text{DATA_TYPE} \\
<\text{shared-targets}> & : = <\text{pf-name}> \\
& \mid <\text{symbol}>
\end{align*}
\]

a. (3 points) How many sentences of the language are possible from this grammar segment? Note: Reason out your answer, just don’t enumerate all of the possibilities!

b. (12 points) Redesign the grammar to remove any questions that don’t make sense. For example, "WHAT ARE LOCAL_VARS OF <symbol>" is such a question.

5. (20 points) Compute FIRST and FOLLOW for each non-terminal in the grammar given below:

\[
\begin{align*}
A & \rightarrow BA' \\
A' & \rightarrow oBA' \mid \epsilon \\
B & \rightarrow D \mid (C) \\
C & \rightarrow DC' \\
C' & \rightarrow aDC' \mid \epsilon \\
D & \rightarrow x \mid nx
\end{align*}
\]

Note that: A, A', B, C, C', and D are non-terminals; a, o, x, n, (, and ) are terminals; \( \epsilon \) stands for the empty symbol.

**Bonus (5 points):** Suppose that the terminals of the grammar are interpreted as: a stands for and, o stands for or, n stands for not, and x stands for relational expressions (i.e., expression relop expression). What language does this grammar represent? Put another way, what kinds of strings does this grammar generate?
Focus on **How/Why** 2 pt. each subpart

1. **SEQS of CIRCLES/ARCS. CIRCLES ARE STATES, ARCS ARE EXPECTED INPUT/OUTPUT ACTIONS**

2. **WHY CAN BE USED TO REPRESENT BOTH TOKEN PATTERNS & SENTENCE STRUCTURE**
   - ONCE DEVELOPED, CAN BE DIRECTLY MAPPED TO ALGORITHM FOR EITHER LEXICAL ANALYSIS OR PARSING.

3. **A TECHNIQUE THAT DISREGARDS/THROWS AWAY TOKENS WHEN ERROR OCCURS UNTIL A SYNC TOKEN IS FOUND**
   - ALLOWS PARSING TO CONTINUE DESPITE ERRORS
   - USED TO READJUST STATE OF PARSER INTO A CONFIG THAT IS CONSISTENT WITH SOME GRAMMAR STATE DERIVATION

4. **FORMAL METHOD FOR SPECIFYING THE PATTERNS OF TOKENS OVER A FIXED ALPHABET, $e, a, ab, a|b, a^*, (a), ...$**
   - LET'S THE TOKEN PATTERNS OF A PROC. LANGUAGE BE DESCRIBED FOR A COMPILER
   - REPS FOR ALL TOKENS CAN SERVE AS INPUT TO A COMPILER-COMPILING TOOL, LIKE LEX, TO AUTOMATICALLY CREATE A LEXICAL ANALYZER.
PATTERN: RULES FOR DESCRIBING STROKES
TOKENS: NAME FOR IDENTIFYING/DISTINGUISHING PATTERNS
LEXEME: AN ACTUAL SEQ OF CHAR FROM INPUT THAT MATCHES PATTERN OF TOKENS

TOKEN LEXEME

SERVES AS BASIS FOR SPECIFYING BOTH THE FUNCTIONALITY & BEHAVIOR OF A LEXICAL ANALYZER

SAME AS HOW FOR C, C++,
P, T, L: COLLECTIVELY ARE USED TO IMPLEMENT LEX ANALYZER

CONSISTS OF NON-TERM, TERMS, PROD RULES, & A START STATE TO SPECIFY THE SYNTAX OF A PROG. LANGUAGE

FORMAL METHOD TO SPECIFY SENTENCE STRUCTURE

ONCE SPECIFIED, A GRAMMAR CAN BE UTILIZED TO CONSTRUCT A PARSER VIA LL(1) OR TABLE-DRIVEN APPROACHES

A TECHNIQUE THAT ATTEMPTS TO RECOVER FROM ERRORS BY CORRECTING INPUT TO HYPOTHESIZE WHAT USER INTENDED

SAME AS FINITE

PARSER DOES MODS TO INPUT: ADDS/DELS TO "CORRECT" TOKENS

TWO KINDS: 1ST & 2ND - BASED ON CORRECT STATE & NEXT INPUT PERFORM ACTIONS TO RECOGNIZE/REJECT STRINGS OVER S

SAME AS C, C++ WHY

REG. EXP. TO DFA CONSTR. BASIS FOR LEXICAL ANALYZER

DFA TO DFA CONVERSION (LED: TOOLS LIKE LEX, JUGI)
#3: \[ S \rightarrow (L) | E_b | a \]
\[ L \rightarrow c L' | (L) L' | E_b L' | a L' \]
\[ L' \rightarrow , S L' | e \]
\[ E \rightarrow [E] E' | c L' c E' | (L) L' c E' | a L' c E' \]
\[ E' \rightarrow d E' | b L' c E' | e \]

#4 a: WHAT <are-is> <what-optics> OF <should-t> 2 4 2
   2 \times 4 \times 2 = 16 \text{ possibilities}

#4 b: <what-what> ::= WHAT <what-what>
   <what-what> ::= ARE <opt-1>
                   \mid IS <opt-2>
   <opt-1> ::= PARAMETERS OR <opt-name>
              \mid LOCALUARS OF <opt-name>
   <opt-2> ::= RETURN TYPE OF <opt-name>
              \mid DATATYPE OF <sym-bol>
#5

\[ \text{First}(A) = \text{First}(B) = \{ \lambda, \eta, c, y \} \]
\[ \text{First}(c) = \text{First}(D) = \{ \lambda, c, y \} \]
\[ \text{First}(c') = \{ \eta, \epsilon \} \]
\[ \text{First}(A') = \{ \emptyset, \epsilon \} \]
\[ \text{Follow}(A) = \text{Follow}(A') = \{ \$, \$ \} \]
\[ \text{Follow}(B) = \{ \$, \$ \} \]
\[ \text{Follow}(c) = \{ \$ \} \]
\[ \text{Follow}(D) = \{ \$, \$, \$, \$ \} \]
\[ \text{Follow}(c') = \{ \$ \} \]