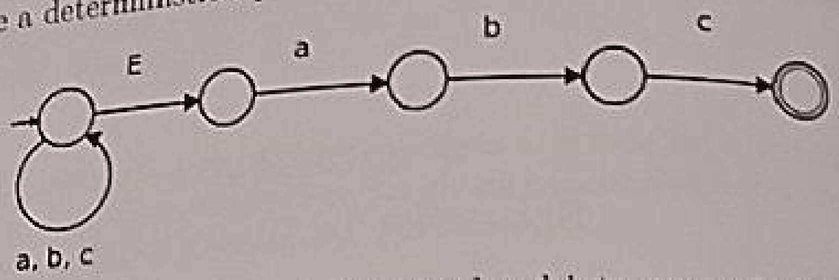
Question 1 *[30%]*

(i) Give (a) a regular expression and (b) a deterministic finite automaton to capture the set of stings over alphabet $\{a, \cdots, z\}$ that contain at most one a and at most one b. *(10%)*

(ii) What language is recognized by the following nondeterminstic finite automaton (NFA)? Give a deterministic equivalent (DFA). Symbol E denotes ϵ below. *(10%)*



(iii) Show how a regular expression may be translated into an equivalent NFA, using $a \cdot (b \mid c) * \cdot d$ to illustrate the technique. *(10%)*

Question 2 *[25%]*

(i) The following are examples of programs written in a simplified Python-like syntax. While pure Python is white-space sensitive, our variant uses special symbols □ (end-line), ▷ (indent) and ◁ (outdent) to indicate program structure explicitly.

```
n = 5 □
fact = −1 □
if n > 0:
  ▷
  fact = n □
  while n > 0:
    ▷
    fact = fact * n □
    n = n −1 □
    ◁
  ◁
ans = fact
```

```
n = 1 □
while n < 100:
  ▷
  n = n + 2 □
  ◁
```

Give a grammar for this mini-language that respects the following rules. A program consists of a sequence of statements, which can be assignment statements, while statements or if statements. Assignments have Java-like syntax but are terminated by an end-line (□) symbol (not semicolon), while multi-line assignments (or conditions) are disallowed. The condition of a while/if is delimited by the keyword and a colon, while the body is delimited by an indent symbol (▷) and an outdent symbol (◁).

You may assume that grammar productions to capture the syntax of expressions have already been written and you may use the non-terminal $\langle exp \rangle$ (that captures expressions) in your grammar. *(15%)*

(ii) Draw a parse tree for the program shown on the right. You need not complete any subtrees representing $\langle exp \rangle$ constructs. *(10%)*

Question 3 *[35 %]*

(i) The grammar shown below is designed to capture the syntax of simplified context-free grammars. Symbols $\Rightarrow$, $\bullet$, $\|$, **N** and **T** are terminals, while nonterminals are shown in the customary $\langle X \rangle$ notation.

$$\begin{aligned}
\langle G \rangle &\rightarrow \langle P \rangle \langle G' \rangle \\
\langle G' \rangle &\rightarrow \langle P \rangle \langle G' \rangle \mid \epsilon \\
\langle P \rangle &\rightarrow \langle L \rangle \Rightarrow \langle R \rangle \bullet \\
\langle L \rangle &\rightarrow \mathbf{N} \\
\langle R \rangle &\rightarrow \langle A \rangle \langle A' \rangle \\
\langle A \rangle &\rightarrow \langle S \rangle \langle S' \rangle \\
\langle A' \rangle &\rightarrow \| \langle A \rangle \langle A' \rangle \mid \epsilon \\
\langle S \rangle &\rightarrow \mathbf{N} \mid \mathbf{T} \\
\langle S' \rangle &\rightarrow \langle S \rangle \langle S' \rangle \mid \epsilon
\end{aligned}$$

Indicate which nonterminals are nullable and the FIRST and FOLLOW sets for each non terminal in the grammar. *(20%)*

(ii) Show the rows in the LL(1) parsing table for these nonterminals: $\langle G \rangle$ and $\langle G' \rangle$. *(10%)*

(iii) Give a crisp description of the LL(1) table-driven, stack-based parsing algorithm. *(5%)*

Question 4 *[10 %]*

Suppose the we wished to augment the mini-language of Question 2 above, with a for loop with the following syntax. (The template on the left illustrates the general syntax, the code snippet on the right is an example that sums the squares of the odd numbers from 1 to 99.)

	total = 0 □
for *var* **in range**(*startval*, *endval*, *stepval*):	**for** n **in range**(1, 100, 2):
▷	▷
body	total = total + n ∗ n □
◁	◁

The intention is that the variable *var* takes on the values *startval*, *startval*+*stepval*, *startval*+ $2\times$ *stepval* $\cdots$ up to but not including *endval* and that the loop body is executed in turn for each such value. Note also that *startval*, *endval* and *stepval* may be expressions ($\langle$expr$\rangle$). Show how a for loop of this kind may be translated into three-address code (TAC), using the code fragment shown on the right to illustrate the technique. Show clearly the correspondence between each element of the source code and the corresponding TAC.