QUIZ 1

Date: 23rd January 2002 — Max. Marks 30

1. Consider the following Lex-like specification of three distinct tokens:

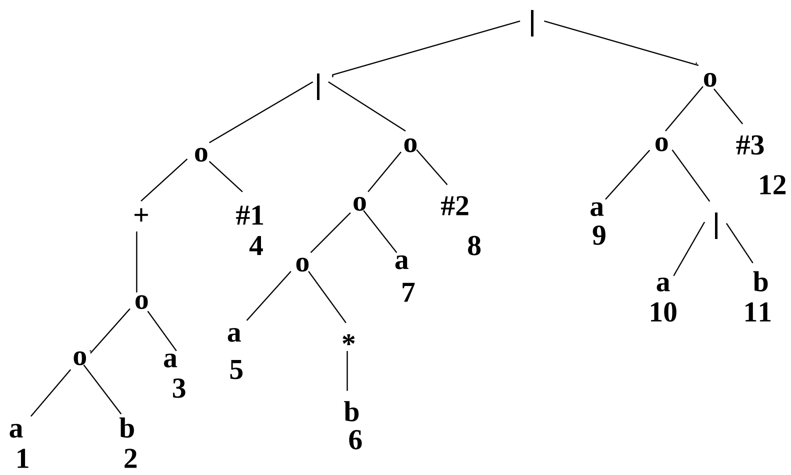
$(aba)+$	Token 1
$(a(b)*a)$	Token 2
$a(a\|b)$	Token 3

   (a) Draw an appropriate syntax tree for the specification which can be directly converted to a DFA. Be careful, because the resulting DFA has to distinguish between lexemes corresponding to the three tokens.

   (b) Number the leaves of the syntax tree from left to right. Now construct the DFA labeling each state with sets of positions.

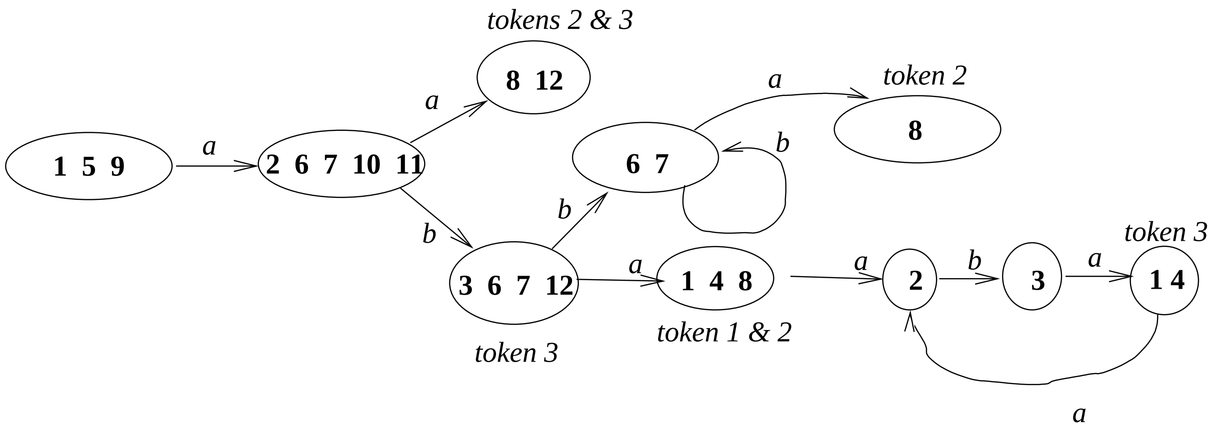
   (c) Indicate the token(s) found against each accepting state. Are there any clashes? If yes, why? (10 Marks)

Answer: The syntax tree shown in the figure below serves our purpose. The three end of string markers #1, #2 and #3 would help us in distinguishing between the three tokens.



2. Number the leaves of the syntax tree from left to right. Now construct the DFA labeling each state with sets of positions.

Answer:



3. Indicate the token(s) found against each accepting state. Are there any clashes? If yes, why?

   Answer: In the figure, accepting states are labeled by the tokens recognised by them. Any state containing position 4 detects token 1. Similarly states containg positions 8 and 12 detect tokens 2 and 3 respectively. One of the clashes is in the state consisting of positions 1, 4 and 8, signifying detection of two tokens, 1 and 2. This is because the two tokens share a common lexeme *aba*.

4. Show that the following grammar, intended to remove the *if-then-else* ambiguity, is still ambiguous.

   stmt → `if expr then` *stmt* | *matched_statement*
   matched_statement → `if expr then` *matched_statement* `else` *stmt*
   | `other` (7 Marks)

   Answer: Consider the sentence `if` *e* `then if` *e* `then` *o* `else if` *e* `then` *o* `else` *o*. Clearly the sentence has two parses indicated by the parentheses

   `if` *e* `then` (`if` *e* `then` *o* `else` (`if` *e* `then` *o* `else` *o*))
   and
   `if` *e* `then` (`if` *e* `then` *o* `else` (`if` *e* `then` *o*)) `else` *o*

   The idea is that a *matched_statement* is really not matched. It can contain a unmatched `then`. If I take such an instance of *matched_statement* and insert it in `if expr then` *matched_statement* `else` *stmt*, the extra `then` can either stand on its own or associate with the `else` to its right.

5. Consider the grammar show below:

   $E \rightarrow E{+}T \mid T$
   $T \rightarrow -T \mid (E) \mid id$

   (a) Remove left recursion from the grammar.

   (b) Write a predictive recursive descent parser for the grammar.

Answer: After removing left recursion, we shall get:

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \ |\ \epsilon$
$T \rightarrow -T \ |\ (E)\ |\ id$

The predictive parser is:

```
function E ()
begin
  if T() then return Eprime() else return false
end

function Eprime()
begin
  if match('+') then
     if T() then return Eprime()
  else return match('$');  // if next symbol is '$' (which is in
end                        // FOLLOW(E')), then use the production
                           // E' ->  epsilon

function T()
begin
  if match('-') then return T()
  else if match('(') then if E() then return match(')')
                          else return false
       else if match(id) then return true
            else return false
end
```

6. Construct the LL(1) parsing table for the grammar shown below. (6 Marks)

$S \rightarrow a\ X\ Y\ b$
$X \rightarrow X\ c \ |\ d$
$Y \rightarrow Z\ W$
$Z \rightarrow b \ |\ \epsilon$
$W \rightarrow e \ |\ \epsilon$

	a	b	c	d	e
S	$S \rightarrow a\ X\ Y\ b$				
X				$X \rightarrow X\ c \ \|\ d$	
Y		Y $\rightarrow$ Z W			$Y \rightarrow Z\ W$
Z		$Z \rightarrow b \ \|\ \epsilon$			$Z \rightarrow \epsilon$
W		$W \rightarrow \epsilon$			$W \rightarrow e$

For what value n is the grammar LL(n)? Explain your answer.

There is no such value. Consider the string $dccc$ generated by X. To parse this string, the production $X \to X\ c$ has to be applied thrice, followed by an application of $X \to d$. Even if we assumed a lookahead of 4, the parser is not equipped with the logic to count the number of times it has used a production.

Augmenting the parser with such a logic will also not help, since a lookahead of n will not suffice for a string $dcc \dots cc$ of length $n + 1$.