Compilers 247. Exam.

Section A.

1. (a) $\Sigma = \{a, b\}$

$L = \{ s \mid s \text{ contains } 3k \text{ as. } k>0 \ k \in \mathbb{Z}^+ \}$.

(b) abbaaa. False.

abbaaa = a(bb)(aa)a. has even number of as

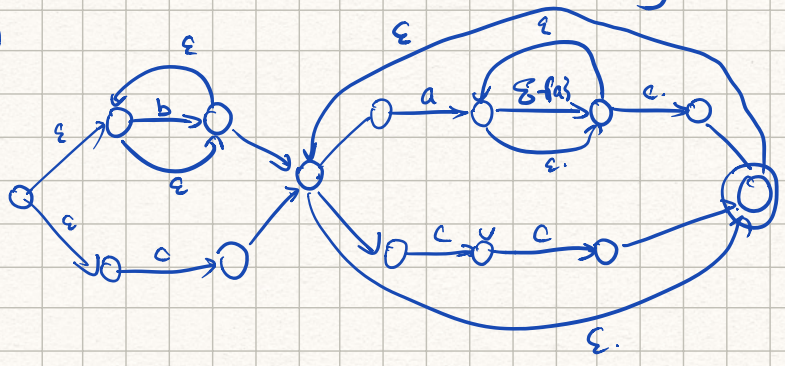
(c) True.

Jflex is a lexer generator. (if you mean .lex file, jflex accepts Contextual languages)

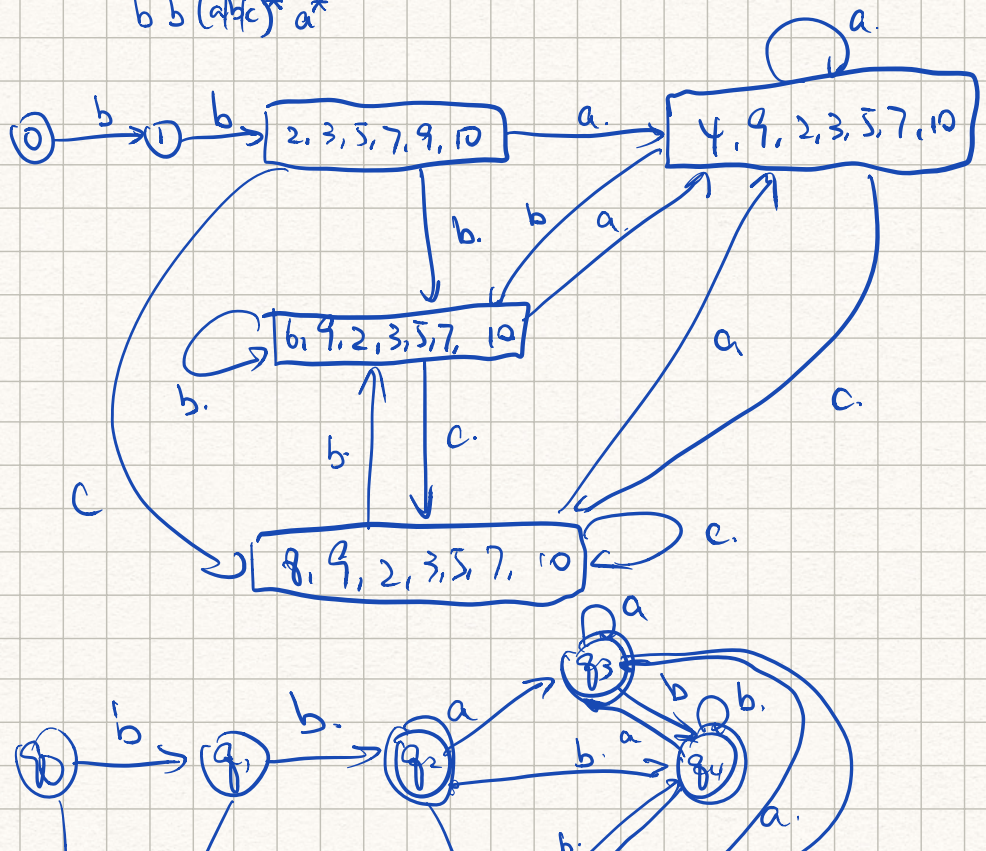
lexer only recognises regex $\leftrightarrow$ RL.

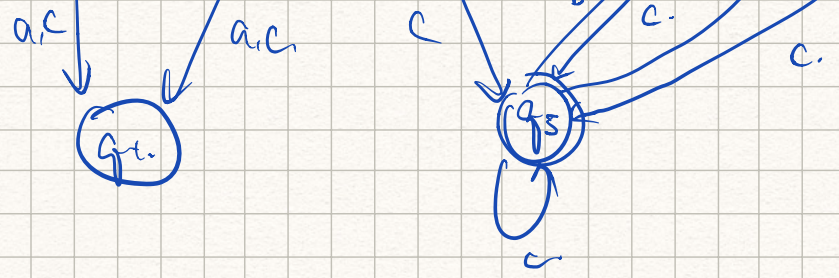
lexer does is only regex pattern matching $\rightarrow$ Token stream.

(d)



(e) bb(a|c)*a*





2. (a) Benefits: do not have to move data around when optimising. only use pointers. (triples).

Saves space (not redundant tmps). (quadruples).

(b) Inlining: putting functions codes into the place where the fuction is called.

Risk. if fuction is large, code size ↑↑↑.

Constant Folding: where constants are unknown, do the calculation at compile time

Risk. be aware of the precision.
compile time precision should = run time precision.

(c)

Function call location.

parameters

local variables / tmp variables

return value.

access link.

control link / ptr to previous AR.

(d) Assembly → IR.

```
T(expr1).
Loop:
    t = T(expr2).
    fjump t END
    T(expr3).
    jump Loop
END
```

(e) `if(n==1) return 1.`
`return foo(n-1) x n.`

Calculates factorial for $n \geq 1$.

```
int foo (int n) {
    if (n==1) return 1;
    return n x foo(n-1)
}
```

}.

Section B

3. (a).

$S \to Sa \mid Xx \mid y$
$X \to z \mid Sd.$
$Y \to Yy \mid d \mid \varepsilon.$

$S \to Xx \mid y \mid aS'$
$S' \to aS' \mid \varepsilon.$
$X \to z \mid Sd.$
$Y \to d \mid \varepsilon \mid yY'$
$Y' \to yY' \mid \varepsilon.$

(b)

i.

	First	Follow
S	x.	\$ ∪ Follow(B) = \$, x.
A	F(B) ∪ F(D) = x, y, z, ε.	\$ ∪ Follow(S) = \$, x.
B	x ∪ F(C) = x, y	x
C	y, ε.	x
D	z, ε.	Follow(B) ∪ Follow(A) ∪ Follow(C) ∪ Follow(D) = \$, x.

ii.

	x	y	z	\$.
S	S→xA.			
A	A→Bx / A→D.	A→Bx.	A→D	A→D
B	B→xS / B→GD.	B→GxD		
C	C→ε.	C→yD.		
D	D→ε.		D→zD	D→ε.

(c)

$S \to E$
$E \to (E) \mid A \mid B$
$A \to A+A \mid A*A \mid t.$
$B \to \neg B \mid B \vee B \mid t.$

$S \to \cdot E, \$$
$E \to \cdot (E), \$$
$E \to \cdot A, \$$
$E \to \cdot B, \$$

$A \to \cdot A+A, \$, +$
$A \to \cdot A*A, \$, *$
$A \to \cdot t, \$$
$B \to \cdot \neg B, \$$
$B \to \cdot B \vee B, \$, \vee$
$B \to \cdot t, \$$

(d). core remains the same.

merge same core, but different lookahead items

might cause more reduce/reduce conflict, since lookahead are merged.

saves a lot more nodes/states

4. (a) Semantic analysis - type checking

int x;

x = 'a'.
EXPR.

- scope checking

context free grammer ← parser.
(technically you can).
adds a ton of new grammer.

(b) Static binding. x=10 in A.
y = 15.
return 15 in B()

So print 15.

dynamic binding. x=5 in B binds to x.
y = 5+5 in A()
= 10

∴ print 10

(c)

```
public abstract class Node {
    public Node left, right;
    public Object accept(Visitor v);
}

public class Stmt extends Node {
    public String id;
    public Stmt(String id, Node e) {
        this.id = id;
        left = e;
        right = null
    }
    public Object accept(Visitor v) {
        return v.visit(this);
    }
}

public class SymbolTable {
    Hashtable table;
    public SymbolTable() {
        table = new Hashtable();
    }
    public SymbolTable put(String key, Object value) {
        table.put(key, value);
    }
    public Object get(String key) {
        for (var id : table.keySet()) if (key.equals(id))
        return table.get(key);
    }
}
```

```
public class checker implements visit {
    SymbolTable[] myTable;

    public checker () {
        myTable = new SymbolTable.()
    }

    public Boolean visit ( DeclNode node ) {
        // check if exsist
        // if no add
        // if yes then error
    }
```