MCQ & FITB:

(1) YACC = Yet Another Compiler Compiler.

(2) Lexical & Syntax Analyzer = front end of compiler.

(3) Grammar has > 1 Parse tree = Ambigious

(4) P, Q & R are total no. of states in LR(0), SLR(1) & CLR(1) then P < Q < R

→ Operator grammar has no null prodxn.

→ Top down parser is not applicable for left recursive grammar.

→ Lexical Analysis can eliminate white space.

Compiler	Interpreter
(1) Scans whole code.	(1) Translates one statement at a time
(2) Converts source code → object code	(2) don't Converts but scan it line by line
(3) don't require source code for execution	(3) requires source code for execution.
(4) machine code stored in disk	(4) machine code is not stored anywhere
(5) CPU utilization = high	(5) CPU utilization = less
(6) C, C++, C#	(6) Python, Ruby, Perl
(7) more efficient.	(7) less efficient.
(8) fast execution	(8) slow execution

Cross compiler : Runs on one platform ⟶ Produces executable code on other platform.

eg. Compiler on windows produces .exe code for linux.

Left factoring : used in grammar transformation

why?

to eliminate common prefixes in $\boxed{\text{alternative prodx}^n}$ of a non terminal

Error Handler : resolving & detecting errors.

→ error (detected)

takes action to eliminate it.

Ambigious grammar : can generate more than one $\boxed{\text{parse tree}}$ for i/p string.

→ Can create $\boxed{\text{errors}}$ ∴ left factoring & operator precedence is used

---

$\boxed{\text{Syntax Analysis}}$ = Checks grammar of program

DAG = Directed Acyclic Graph

- LL(1) = TOP DOWN
- LALR > SLR
- Ambig. Grammar can't be LR(1)

LR = left to right & rightmost derivation in reverse

[redacted]

yylex() is automatically generated by FLEX.

A synthesized attribute is a attribute whose value at a parse tree depends on the values of its children in parse tree.

Shift reduce conflict = state don't know if it will make a shift opn or redn for terminal

I/p to Lexical Analyzer = source code being compiled

SDT => ~~[illegible]~~ Syntax Directed Translation.

Augmented Grammar = Additional symbol added to grammar to improve parsing by parser.

S- Attributed Grammar = CFG that associates attribute with grammar symbols & rules

Bottom up

Synthesized attributes

SYNTAX TREE = Tree like DS -> represents syntactic structures of a string

nodes => Parent, children

TOKEN = Group of symbols

PATTERN = sequence of lexeme / set of rules & instruction

Lexeme = sequence of characters in the source program.

Finite Automata is used for recognition of token.

mul = 4*3

→ identifier.

Canonical LR = most powerful parser.

Data items grouped together = Record

Code, Procedures, Variable = managed by Run Time Env.

Lexical Analyzer reads source_program & break it into tokens

LR(1) is look ahead LR Parser.

Inherited $\xrightarrow{gets}$ value from siblings or parent

Synthesized $\xrightarrow{gets}$ value from children to parent

Common subexpression elimination → elimination takes place in intermediate code generation

Recursive Decent Parser = TOP-DOWN Parser

Compiler can check SYNTAX ERRORS

Part of string $\xrightarrow{matches}$ RHS of any prodxn = handle

Semantic Analyzer are tokens together into semantic structure.

↳ grammar checking

pf("i = %d, &i = %x", i, &i); = 21 tokens [∵ exp. is very long]

elimination of left recursion is done via $\boxed{\text{TOP DOWN PARSER}}$

Lexical Analyzer skips white spaces

DAG is used to represents common subexpressions

→ optimizes code by reducing redundancy.

Error recovery by panic mode, phrase level, error prodxn