CD LAB VIVA QUESTIONS

EXPERIMENT 1

1. Explain LEX and YACC tools briefly.
2. Give the structure of the LEX program.
3. Explain the structure of a YACC program.
4. What are tokens or terminal symbols?
5. What is lexical analyzer?
6. Discuss about the input buffering scheme in lexical analyzer.

EXPERIMENT 2

1. What are the limitations of CFG?
2. Explain Ambiguous and Unambiguous Grammar.
3. Which grammar can be translated to DFAs?
4. Differentiate between Ambiguous and Unambiguous Grammar.
5. Determine the language which is generated by the grammar
   $S \rightarrow aSa|bSb|a|b$ over the alphabet of {a,b}

EXPERIMENT 3

1. How many Keywords are in C?
2. What is a C token and types of C tokens?
3. What is an Identifier?
4. How many types of operators are there in C?
5. What is a variable?

EXPERIMENT 4

1. Why is left recursion considered problematic?
2. Give an example of grammar with left recursion and show how to eliminate it.
3. How does left recursion affect parsing efficiency?
4. What is the difference between indirect and direct left recursion?
5. What is a recursion method?
6. Eliminate the left recursion for the following grammer
   E-->E+T/T
   T-->T*F/F
   F-->(E)/id

EXPERIMENT 5

1. Do left factoring in the following grammar-

   $S \rightarrow a / ab / abc / abcd$

2. What is the relationship between left recursion and left factoring?
3. Why is left factoring used?

4. How does left factoring help in parsing?
5. Provide an example of left factoring.

EXPERIMENT 6

1. When is the type checking usually done? (duringSDT)
2. Who checks every character of the source text in a Compiler? (LA)

For questions 3 and 4 refer to the data given below:

The programming language given below is written in the programming language that does not allow nested declarations of functions and allows global variables.

```
global int j = 100, k = 5;
void M(n)
{
int j = 10;
print (n + 10);
j = 200;
k = 20;
print (n);
}
main()
{
M(j + k);
}
```

3. What is the output of the above program if the programming language uses static scoping and call by need parameter passing mechanism?
4. What is the output of the above program if the programming language uses dynamic scoping and call by name parameter passing mechanism?

EXPERIMENT 7

5. What is Bottom up parsing?
6. What do you mean by Top Down Parsing?
7. Differentiate between top down and bottom up parsing?
8. What is the leading of a non-terminal?
9. What is the trailing of a non-terminal
10. What are the techniques used in Bottom up Parsing?
11. Which parsing is more beneficial?

EXPERIMENT 8

1. What are four actions of Shift Reduce Parser?
2. Shift-reduce parsing is a form of bottom-up parsing. What is the purpose of the two actions, shift and reduce?
3. Describe the stack implementation of shift reduce parsing?
4. The Earley algorithm is an efficient context-free parsing algorithm which uses a chart data structure. Is this algorithm a top-down or a bottom-up algorithm?
5. State Error in each phase of compiling?

EXPERIMENT 9

1. Distinguish between FIRST and FOLLOW?
2. What is the significance of first & follow?
3. What do you mean by handle pruning?
4. What are the techniques used in Top Down Parsing?
5. How many rules are there to calculate First & Follow?
6. Given following grammar
   S -> L=L
   s -> L
   L -> *L
   L -> id
   What is the first and follow for the non-terminals?
7. If the grammar is changed into S -> L=R
   S -> R
   L -> *R
   L -> id
   R -> L
   What will be the first and follow?

EXPERIMENT 10

1. What do you mean by CFG?
2. What do you mean by operator precedence?
3. What is the property of Operator Precedence Grammar?
4. Why we require Operator Precedence Grammar?
5. What are Precedence Relations in Operator Grammar?
6. What is C Operator Precedence and Associativity?
7. What are Precedence Functions in compiler design?

Viva Questions(Program-1)

Question 1:

What is LEX?

Answer:

LEX is a lexical analyzer generator that transforms a set of regular expressions into a C program that performs lexical analysis.

Question 2:

What is YACC?

Answer:

YACC stands for "Yet Another Compiler Compiler." It generates a parser that converts a sequence of tokens into a syntax tree according to a grammar.

Question 3:

How do LEX and YACC work together?

Answer:

LEX is used to tokenize the input, while YACC processes these tokens to analyze the structure using a specified grammar.

Question 4:

What is a token in LEX?

Answer:

A token is a categorized block of text, such as keywords, identifiers, or operators, recognized during lexical analysis.

Question 5:

Can YACC handle ambiguous grammars?

Answer:

Yes, but it requires additional techniques such as precedence rules or associativity to resolve conflicts.

Viva Questions(Program-2)

Question 1:

What is a grammar in compiler design?

Answer:

A grammar is a set of production rules that describe the syntactical structure of a language.

Question 2:

What are the components of a grammar?

Answer:

A grammar consists of terminals, non-terminals, a start symbol, and production rules.

Question 3:

How does one check if a string belongs to a grammar?

Answer:

By parsing the string according to the grammar's production rules using techniques like top-down or bottom-up parsing.

Question 4:

What is top-down parsing?

Answer:

Top-down parsing starts from the start symbol and attempts to derive the string by applying production rules.

Question 5:

What role do non-terminals play in a grammar?

Answer:

Non-terminals represent intermediate symbols that can be replaced by terminals or other non-terminals according to the production rules.

Viva Questions(Program-3)

Question 1:

What is a keyword in a programming language?

Answer:

A keyword is a reserved word in a programming language that has a predefined meaning and cannot be used as an identifier.

Question 2:

How can a string be checked for keywords?

Answer:

By comparing each word in the string with a list of reserved keywords.

Question 3:

Why is keyword recognition important in compilers?

Answer:

Keywords often represent control structures and data types, so recognizing them is essential for correct syntax analysis.

Question 4:

Can a keyword be used as a variable name?

Answer:

No, keywords are reserved and cannot be used as identifiers in the source code.

Question 5:

What data structure is typically used to store keywords for comparison?

Answer:

Hash tables or arrays are commonly used for fast lookup of keywords.

Viva Questions(Program-4)

Question 1:

What is left recursion?

Answer:

Left recursion occurs when a production rule has the form $A \rightarrow A\alpha$, where A is a non-terminal and α is a string of terminals and non-terminals.

Question 2:

Why is left recursion a problem in parsers?

Answer:

Left recursion can cause infinite recursion in top-down parsers, leading to non-termination.

Question 3:

How do you eliminate left recursion?

Answer:

By transforming the grammar into an equivalent one without left recursion, typically by introducing new production rules.

Question 4:

What is an indirect left recursion?

Answer:

Indirect left recursion occurs when a non-terminal indirectly recurses through another non-terminal, such as $A \rightarrow B\alpha$ and $B \rightarrow A\beta$.

Question 5:

Can all left-recursive grammars be converted to non-left-recursive grammars?

Answer:

Yes, all left-recursive grammars can be rewritten to eliminate left recursion.

Viva Questions(Program-5)

Question 1:

What is left factoring?

Answer:

Left factoring is a technique used to remove ambiguity by factoring out the common prefixes of the production rules in a grammar.

Question 2:

Why is left factoring needed?

Answer:

It simplifies the parsing process by reducing the number of choices at each step, particularly in top-down parsers.

Question 3:

How do you perform left factoring on a grammar?

Answer:

By extracting the common prefixes from the production rules and introducing new non-terminals.

Question 4:

What is the result of applying left factoring?

Answer:

The grammar becomes easier to parse, reducing ambiguity and improving efficiency.

Question 5:

Can left factoring be applied to any grammar?

Answer:

Yes, left factoring can be applied to any grammar that has common prefixes in its production rules.

Viva Questions(Program-6)

Question 1:

What is the purpose of YACC in parsing arithmetic expressions?

Answer:

YACC generates a parser that builds a syntax tree for arithmetic expressions based on grammar rules.

Question 2:

What are the precedence rules in YACC?

Answer:

Precedence rules in YACC determine the order of evaluation for operators, helping resolve conflicts between operators like + and *.

Question 3:

How do you define grammar rules in YACC?

Answer:

Grammar rules in YACC are defined in the %% section, specifying how tokens are combined to form valid expressions.

Question 4:

What is an action in a YACC rule?

Answer:

An action is a block of C code associated with a grammar rule that is executed when the rule is matched during parsing.

Question 5:

Can YACC handle both integer and floating-point arithmetic?

Answer:

Yes, YACC can handle multiple types of arithmetic by defining appropriate tokens and actions.

Viva Questions(Program-7)

Question 1:

How do LEX and YACC interact in a calculator program?

Answer:

LEX scans the input to tokenize numbers and operators, and YACC parses the tokens and evaluates the expression.

Question 2:

What are shift and reduce operations in YACC?

Answer:

“Shift” moves a token onto the stack, while “reduce” replaces a sequence of tokens with a non-terminal according to a production rule.

Question 3:

How do you handle errors in a LEX/YACC calculator?

Answer:

You can define an error rule in YACC and use yyerror() to handle syntax errors and provide feedback.

Question 4:

What is a semantic action in YACC?

Answer:

A semantic action is C code embedded in a YACC rule that performs calculations or stores values during parsing.

Question 5:

What type of arithmetic expressions can this calculator handle?

Answer:

It can handle basic expressions involving addition, subtraction, multiplication, and division.

Viva Questions(Program-8)

Question 1:

What is a recursive descent parser?

Answer:

A recursive descent parser is a top-down parser where each non-terminal in the grammar is represented by a recursive function.

Question 2:

How does a recursive descent parser work?

Answer:

It starts from the start symbol and makes recursive calls for each production rule, parsing the input one symbol at a time.

Question 3:

What kind of grammar is suitable for recursive descent parsing?

Answer:

Recursive descent parsing works well with grammars that are non-left-recursive and LL(1), meaning they can be parsed with one lookahead symbol.

Question 4:

What is a lookahead in recursive descent parsing?

Answer:

A lookahead is the next input symbol that the parser uses to decide which production rule to apply.

Question 5:

Can recursive descent parsers handle ambiguous grammars?

Answer:

No, recursive descent parsers cannot handle ambiguous grammars directly. The grammar must be transformed to remove ambiguity.

Viva Questions(Program-9)

Question 1:

What are the basic operations of a stack?

Answer:

The basic operations are push, pop, peek (or top), and isEmpty.

Question 2:

What is a stack used for in a compiler?

Answer:

Stacks are used for managing function calls, expression evaluation, and syntax parsing in compilers.

Question 3:

How does the push operation work?

Answer:

Push adds an element to the top of the stack.

Question 4:

What does the pop operation do?

Answer:

Pop removes and returns the element at the top of the stack.

Question 5:

What is the significance of the peek operation?

Answer:

Peek allows viewing the top element of the stack without removing it, useful for decision-making during parsing.

Viva Questions(Program-10)

Question 1:

What is the leading of a non-terminal?

Answer:

The leading of a non-terminal is the set of terminals that can appear at the beginning of a string derived from that non-terminal.

Question 2:

Why is it important to calculate the leading of non-terminals?

Answer:

It is important for determining which production rule to apply during parsing.

Question 3:

How do you calculate the leading of a non-terminal?

Answer:

By examining the production rules and identifying the terminals that can appear first in each rule.

Question 4:

What is the role of leading sets in predictive parsers?

Answer:

Leading sets help predictive parsers make decisions about which production rule to use based on the current input token.

Question 5:

What is the difference between leading and first sets?

Answer:

Leading refers to the terminals that appear at the start of a derivation, while first is used to refer to the first terminal that can be derived from a non-terminal.

Viva Questions(Program-11)

Question 1:

What is shift-reduce parsing?

Answer:

Shift-reduce parsing is a bottom-up parsing technique that uses a stack to shift input tokens and reduce them according to production rules.

Question 2:

What is the “shift” operation in shift-reduce parsing?

Answer:

The “shift” operation moves the next input symbol onto the parsing stack.

Question 3:

What is the “reduce” operation in shift-reduce parsing?

Answer:

The “reduce” operation replaces a sequence of symbols on the stack with a non-terminal, based on a production rule.

Question 4:

What is a handle in shift-reduce parsing?

Answer:

A handle is the substring that matches the right-hand side of a production rule, which is then reduced to a non-terminal.

Question 5:

What are the common types of conflicts in shift-reduce parsing?

Answer:

Shift-reduce conflicts and reduce-reduce conflicts occur when the parser cannot decide whether to shift or reduce.

Viva Questions(Program-12)

Question 1:

What is the FIRST set in a grammar?

Answer:

The FIRST set of a non-terminal contains the terminals that can appear at the beginning of a string derived from that non-terminal.

Question 2:

How is the FIRST set used in predictive parsing?

Answer:

The FIRST set helps the parser determine which production rule to apply based on the next input symbol.

Question 3:

How do you calculate the FIRST set for a non-terminal?

Answer:

By recursively analyzing the production rules and collecting the first terminal symbols that appear in each derivation.

Question 4:

Can ε (epsilon) be part of the FIRST set?

Answer:

Yes, if a non-terminal can derive an empty string (ε), then ε is included in its FIRST set.

Question 5:

How do you handle FIRST sets for non-terminals that start with other non-terminals?

Answer:

The FIRST set of the non-terminal includes the FIRST set of the non-terminal that appears at the beginning of its production rules.

Viva Questions(Program-13)

Question 1:

What is an operator-precedence grammar?

Answer:

An operator-precedence grammar allows the parser to determine the precedence of operators without needing to transform the grammar.

Question 2:

What are precedence relations in an operator-precedence grammar?

Answer:

Precedence relations define whether one operator has higher, lower, or equal precedence compared to another.

Question 3:

How do you check if a grammar is operator-precedence?

Answer:

By constructing the precedence relations between terminals and ensuring that the grammar has no ambiguity or conflicts.

Question 4:

What is the difference between a shift-reduce parser and an operator-precedence parser?

Answer:

An operator-precedence parser specifically uses precedence relations to resolve conflicts, while shift-reduce parsers rely on general production rules.

Question 5:

Can all grammars be transformed into operator-precedence grammars?

Answer:

No, only grammars that do not contain ambiguities or certain types of recursion can be transformed into operator-precedence grammars.

Viva Questions(Program-14)

Question 1:

What is intermediate code in compiler design?

Answer:

Intermediate code is a low-level, machine-independent representation of the source program, typically generated between the source code and target code.

Question 2:

Why do we generate intermediate code?

Answer:

Intermediate code simplifies optimization and allows compilers to target multiple machine architectures.

Question 3:

What is three-address code?

Answer:

Three-address code is a type of intermediate code where each instruction contains at most three operands, typically for binary operations.

Question 4:

What is a quadruple in intermediate code?

Answer:

A quadruple is a representation of an intermediate code instruction with four fields: operator, argument1, argument2, and result.

Question 5:

How do you generate intermediate code for an arithmetic expression?

Answer:

By breaking the expression into smaller parts and assigning temporary variables to store intermediate results.

Viva Questions(Program-15)

Question 1:

What is DFA minimization?

Answer:

DFA minimization is the process of reducing the number of states in a deterministic finite automaton (DFA) while preserving its language.

Question 2:

Why is DFA minimization important?

Answer:

It simplifies the DFA, making it more efficient in terms of time and space during execution.

Question 3:

How do you minimize a DFA?

Answer:

By identifying and merging equivalent states, typically using techniques like partition refinement.

Question 4:

What are equivalent states in a DFA?

Answer:

Equivalent states are states that behave identically for all possible input strings, leading to the same set of transitions.

Question 5:

Can a minimized DFA be further reduced?

Answer:

No, once a DFA is minimized, it cannot be further reduced without altering the language it accepts.

Viva Questions(Program-16)

Question 1:

What is the difference between an NFA and a DFA?

Answer:

An NFA (Non-deterministic Finite Automaton) allows multiple transitions for the same input symbol or ε-transitions, while a DFA (Deterministic Finite Automaton) has exactly one transition per input symbol.

Question 2:

What is the subset construction method?

Answer:

The subset construction method is used to convert an NFA into an equivalent DFA by creating states in the DFA that represent sets of NFA states.

Question 3:

Why do we convert an NFA to a DFA?

Answer:

A DFA is easier to implement as it guarantees a single unique transition for each input symbol, which simplifies pattern recognition.

Question 4:

Does every NFA have an equivalent DFA?

Answer:

Yes, every NFA can be converted into an equivalent DFA that accepts the same language.

Question 5:

How does the state explosion problem affect NFA to DFA conversion?

Answer:

The number of states in the resulting DFA can grow exponentially compared to the NFA, leading to the state explosion problem.