2022.pdf 2021.pdf 2019.pdf 2017.pdf

2022

Group 1:

1. What is lexical analyzer in Compiler Design? With examples explain (i) patterns, (ii) lexemes, and (iii) tokens.
>
In compiler design, a lexical analyzer (also known as a lexer or scanner) is a program or module that reads the input source code and divides it into a sequence of tokens or lexemes, which are meaningful units such as keywords, identifiers, operators, and literals

_

Int x = 5;

LEXEME	PATTERN	TOKEN
Int	int | float | …	type
X	\w+	id
=	=|>|<|...	op
5	\d+	integer_constant
;	;	delimiter

2. (a) Based on Chomsky's hierarchy, discuss the difference between Context free grammar and Context sensitive grammar.

Regular language : grammar of form A -> Ba for all production or A -> aB for all production. Accepted by DFA.
eg -> {S -> 0S | 0}. Used in lexical analysis phase of compiler where lexemes are accepted by DFA.

CFL : grammar of form A -> α, α is string of terminals and non-terminals. Accepted by PDA.
eg -> {S -> 0S1 | 01}. Used in syntax analysis phase to check syntax of language.

CSL : grammar of form α->β, |α|>=|β|. Accepted by LBA.
eg -> {S -> ASBC | ABC, A -> 0, CB -> BC, B -> 1, C -> 2}.
Used in semantic analysis of compiler to check semantics of language

Or

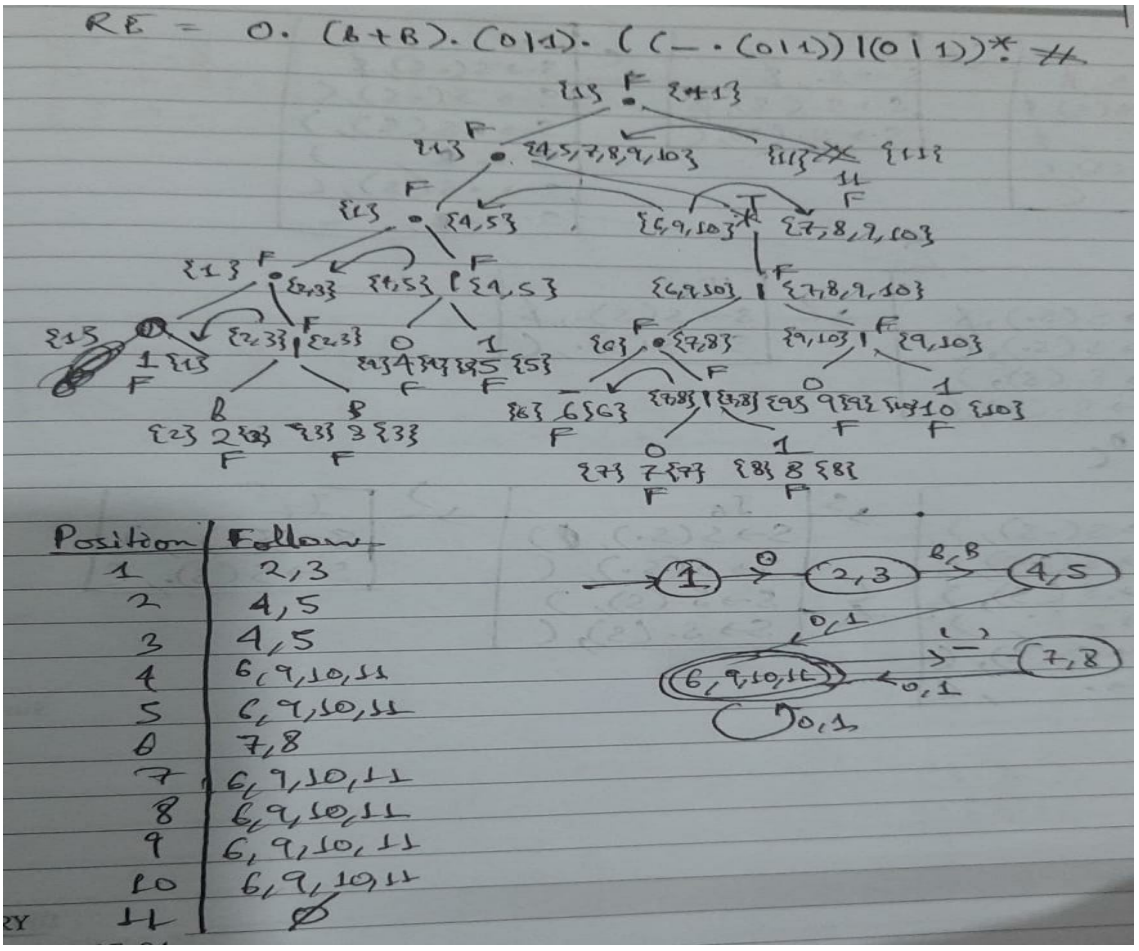
Context-free grammars (CFGs) are a type of formal grammar in which each rule consists of a nonterminal symbol on the left-hand side and a sequence of zero or more terminal and nonterminal symbols on the right-hand side. The rules are applied in a top-down fashion, starting with the start symbol and repeatedly replacing nonterminals with their corresponding right-hand side expansions until only terminal symbols remain. The resulting strings are said to be in the language generated by the CFG.

One of the main limitations of CFGs is that they cannot express certain types of linguistic phenomena that depend on context. For example, they cannot enforce agreement between the subject and verb in a sentence, or ensure that an open parenthesis is always matched with a closing parenthesis. Context-sensitive grammars (CSGs) are a more powerful type of grammar that can handle these types of phenomena.

In a CSG, each rule has the form αAβ → αγβ, where A is a nonterminal symbol, α and β are sequences of terminal and nonterminal symbols, and γ is a nonempty sequence of symbols such that |γ| ≤ |A|. The idea is that the rule can only be applied if the substring A appears in the context of α and β. This allows CSGs to enforce context-dependent constraints on the generated strings.

—

A binary constant inC starts with "0b" or "0B' followed by a sequence of Os and Is. One "_" can b 0b00101100; 0b0001_1000; 0B0_1_0_1_0_1_0_1_0_1_0_1_0_1_0_1. e written between numbers.
Examples: Write a regular expression for the binary constant. Construct a DFA directly from the regular expression



Group 2:

3 (a) While constructing a parsing table, why do you need to construct the FIRST and FOLLOW sets? Explain your answer with examples.

The FIRST and FOLLOW sets are used in the construction of predictive parsing tables, which are used by top-down parsers to parse the input string. Predictive parsing tables are based on LL(1) grammars, which are context-free grammars that satisfy a number of conditions, including the property that no two production rules for the same nonterminal have the same FIRST set.

or,

First set of a non-terminal can tell us *whether there is a production rule for that non-terminal that can produce the terminal at the current head of input,* then we can apply the particular production rule to derive the current terminal.

Follow set tells us *whether the current non terminal will be arrived at later down the line while parsing by reducing the current non-terminal to epsilon or reducing it according to some other production rule.*

4 (d) What are viable prefixes? Show at least two viable prefixes in the grammar.

A viable prefix is a prefix of a right sentential form that can appear during bottom up parsing.

In bottom-up parsing, a viable prefix is a prefix of the input string that could be the beginning of a valid right-hand side of a production in the grammar being parsed. In other words, a viable prefix is a sequence of input symbols that can be reduced to a nonterminal symbol using a production rule of the grammar.

5. a) What is a handle in shift-reduce parsing? What is handle pruning?

In shift-reduce parsing, a handle is a substring of the input string that matches the right-hand side of a production in the grammar and can be reduced to the corresponding non-terminal symbol.

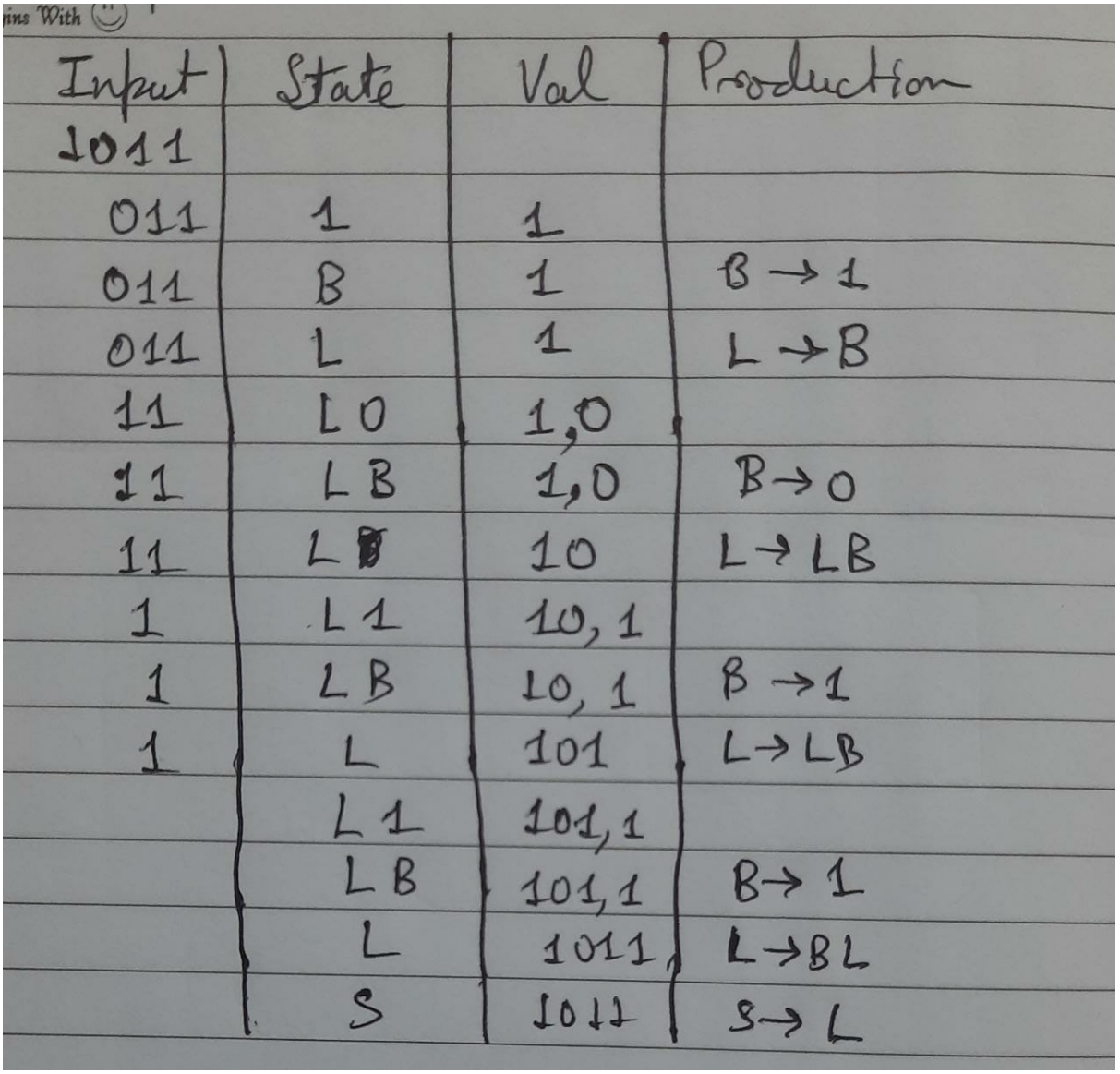
Handle pruning is a technique that helps to improve the efficiency of shift-reduce parsing algorithms by reducing the size of the stack and avoiding unnecessary reduce actions. It involves checking whether the sequence of tokens on the top of the stack form a handle or not. If the sequence is not a handle, then it cannot be reduced, and we can safely remove it from the stack.

Group 3:
6. (a) Define S-attributed and L-attributed grammar. Why are they important in semantic analysis phase of a compiler?

(b) The following grammar generates binary numbers:
S -> L
L -> LB I B
B -> 0 |1
Design an S-attributed definition SDD to compute S.val, the decimal-number value of an Write a input string. translation scheme for the above SDD.
(c) Using a bottom-up parser, show how you can evaluate the decimal value of the binary string 1011.

An S-Attributed Definition is a Syntax Directed Definition that uses only synthesized attributes.
L-Attributed Definitions : A grammar is L-attributed if each attribute a_j at X_i of a grammar rule:
X0-> X1 X2 ... Xn
1) is either a synthesized attribute, or
2) the value of a_j at X_i only depends on attribute of the symbols X0, ... Xi-1, that occur to the left of X_i in the grammar rule, or
3) depends on the inherited attributes of X0
.
S-attribute and L-attribute are used to implement translation schemes for production rules.

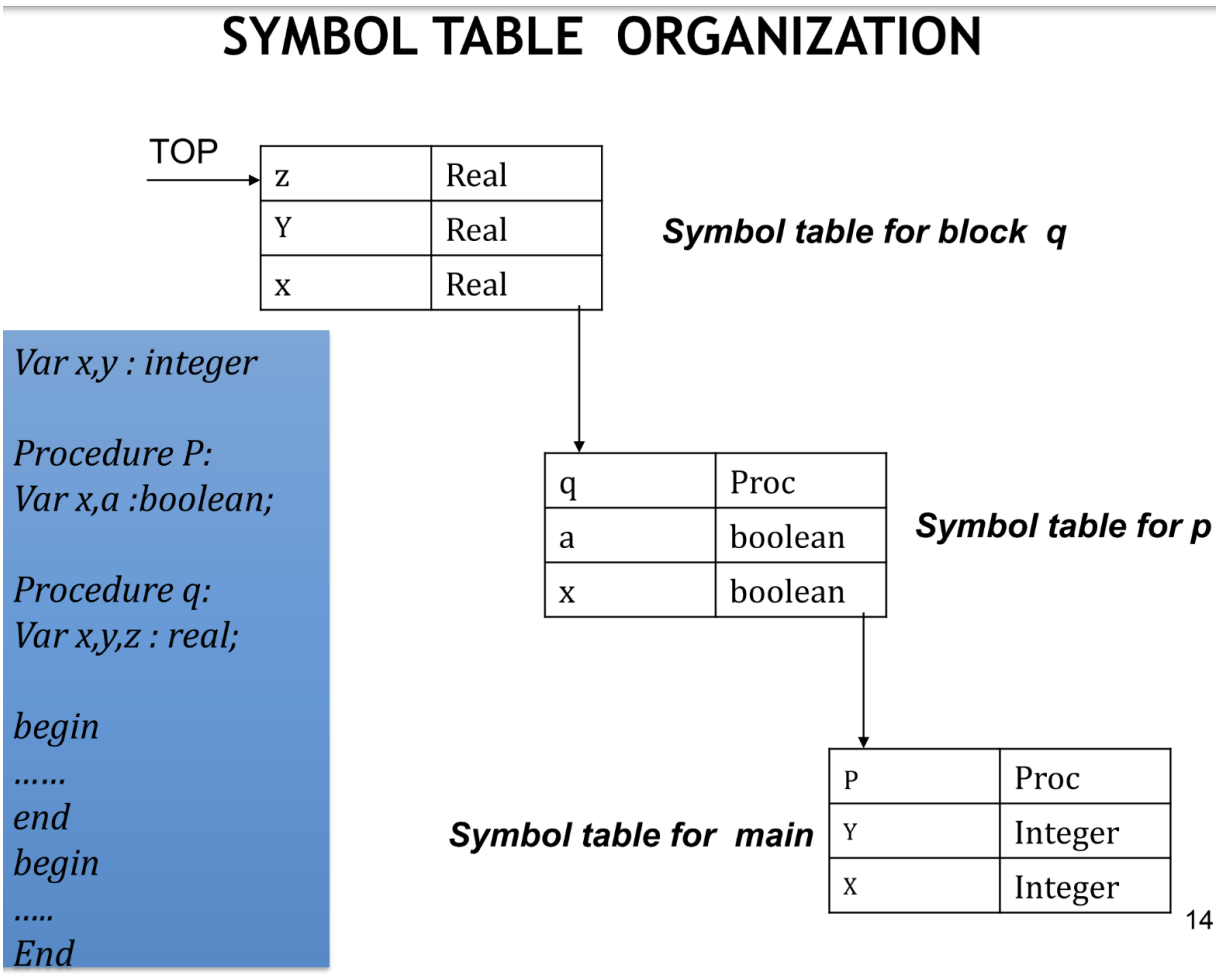
Production	Semantic rule
S -> L	S.val = L.val
L -> L1 B	L.val = (L1.val << 1) + B.val
L -> B	L.val = B.val
B -> 0	B.val = 0
B -> 1	B.val = 1



Input	State	Val	Production
1011			
011	1	1	
011	B	1	B → 1
011	L	1	L → B
11	L 0	1,0	
11	L B	1,0	B → 0
11	L [illegible]	10	L → LB
1	L 1	10, 1	
1	L B	10, 1	B → 1
1	L	101	L → LB
	L 1	101, 1	
	L B	101, 1	B → 1
	L	1011	L → BL
	S	1011	S → L

7. (a) What type of information is stored in the symbol table ? Discuss how the scope rules are stored in a symbol table implemented as multiple hash table ?

- Examples:
  - Variables and Constants
    - Type, line number where declared, lines where referenced, scope
  - Procedure or Function
    - Number of parameters, parameters themselves, result type
  - Array
    - Dimensions, array bounds



7. (c) What is three-address code? Discuss different implementations of three-address code.

Checkout this link - https://www.geeksforgeeks.org/three-address-code-compiler/

Group 4:

10. (b) Briefly explain how "Graph Coloring algorithm' can be used for register allocation.

8.8.4 Register Allocation by Graph Coloring

When a register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored (*spilled*) into a memory location in order to free up a register. Graph coloring is a simple, systematic technique for allocating registers and managing register spills.

In the method, two passes are used. In the first, target-machine instructions are selected as though there are an infinite number of symbolic registers; in effect, names used in the intermediate code become names of registers and

the three-address instructions become machine-language instructions. If access to variables requires instructions that use stack pointers, display pointers, base registers, or other quantities that assist access, then we assume that these quantities are held in registers reserved for each purpose. Normally, their use is directly translatable into an access mode for an address mentioned in a machine instruction. If access is more complex, the access must be broken into several machine instructions, and a temporary symbolic register (or several) may need to be created.

Once the instructions have been selected, a second pass assigns physical registers to symbolic ones. The goal is to find an assignment that minimizes the cost of spills.

In the second pass, for each procedure a *register-interference graph* is constructed in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where the other is defined. For example, a register-interference graph for Fig. 8.17 would have nodes for names `a` and `d`. In block B_1, `a` is live at the second statement, which defines `d`; therefore, in the graph there would be an edge between the nodes for `a` and `d`.

An attempt is made to color the register-interference graph using k colors, where k is the number of assignable registers. A graph is said to be *colored* if each node has been assigned a color in such a way that no two adjacent nodes have the same color. A color represents a register, and the color makes sure that no two symbolic registers that can interfere with each other are assigned the same physical register.

Although the problem of determining whether a graph is k-colorable is NP-complete in general, the following heuristic technique can usually be used to do the coloring quickly in practice. Suppose a node n in a graph G has fewer than k neighbors (nodes connected to n by an edge). Remove n and its edges from G to obtain a graph G'. A k-coloring of G' can be extended to a k-coloring of G by assigning n a color not assigned to any of its neighbors.

By repeatedly eliminating nodes having fewer than k edges from the register-interference graph, either we obtain the empty graph, in which case we can produce a k-coloring for the original graph by coloring the nodes in the reverse order in which they were removed, or we obtain a graph in which each node has k or more adjacent nodes. In the latter case a k-coloring is no longer possible. At this point a node is spilled by introducing code to store and reload the register. Chaitin has devised several heuristics for choosing the node to spill. A general rule is to avoid introducing spill code into inner loops.

2021

Group 1

1. With examples explain the differences between regular languages, context free languages and context sensitive languages. With reference to the different phases of compilers, discuss which types of languages are used in compiler design.

Group 2

3. (a) While constructing a parsing table, why do you need to construct the FIRST and FOLLOW sets? Explain your answer with examples. -> Repeated

OR

With an example explain how you remove indirect left recursion in a grammar. -> DIY

(b) Discuss how the lookahead is used in LR(1) parsing (Definition is not required. Explain with an example). -> DIY

OR

What is a viable prefix? Explain with an example.-> Repeated

Group 3

8. The following grammar generates binary numbers:

S → L { S.val = L.val }
L → L B | B { L.val = 2*L1.val + B.val } | { L.val = B.val }
B → 0 | 1 { B.val = 0.lexval } | { B.val = 1.lexval }

Design an S-attributed definition SDD to compute S.val, the decimal-number value of an input string. Write a
translation scheme for the above SDD.
Discuss how you evaluate SDD using bottom-up parsers?

9. (a) What is the use of symbol table? In which phases of compiler a symbol table is used?

Symbol table is used to lookup the attributes like address, type of a variable. The symbol table is used during the semantic analysis, IR generation, code optimization and code generation phases.

(b) What are the operations performed on a symbol table when it is implemented as a single hash table?

On scope entry, process declaration, process use and on scope exit.

(c) What is a three-address code? Write a three address code for the following expression: a[i] = b * c + b * d

Three-address code is a low-level code representation used in compilers to represent expressions in a way that can be easily converted into machine code. It is called three-address code because each instruction has at most three operands.

Here's the three-address code for the expression "a[i] = b * c + b * d":

```
t1 = b * c
t2 = b * d
t3 = t1 + t2
t4 = sizeOf(a)
t5 = length(a)
t6 = t4 / t5
t7 = i * t6
t8 = addr(a)
t9 = t8[t7]
t9 = t3
```

In the above code, `t1`, `t2`, and `t3` are temporary variables used to store the intermediate results of the multiplication and addition operations. `t4` is used to calculate the offset of the array `a` based on the index `i` and the size of each element in the array. Finally, `t5` is used to calculate the address of the element `a[i]` and then the value `t3` is stored at that address using the dereference operator `*`.

Group 4

11. (a) With an example explain what are ‘liveness’ and ‘next use’ of variables. Why do you need this information?

In a compiler, liveness analysis is used to determine when a variable is live, i.e., when it is still being used in the program, and when it becomes dead, i.e., when it is no longer being used. Next-use information is related to liveness analysis and is used to determine the next point in the program where a variable will be used or modified after a given point.

For example, consider the following code snippet:

```
x = y + z
y = x * 2
z = y + 1
```

In the above example, the liveness and next-use information for each variable would be as follows:

```
Variable   | Liveness           | Next use
-----------------------------------------
x          | 1st stmt - end     | 2nd stmt
y          | 1st stmt - end     | 3rd stmt
```

z | 1st stmt - 3rd stmt | None
```

With liveness analysis and next-use information, the compiler can determine the set of live variables at each point in the program and can allocate registers or memory locations accordingly. This information is important for optimizing code and minimizing the use of registers and memory.

**(c) Discuss the use of 'Register Descriptor' and 'Address Descriptor'.**

### 8.6.1 Register and Address Descriptors

Our code-generation algorithm considers each three-address instruction in turn and decides what loads are necessary to get the needed operands into registers. After generating the loads, it generates the operation itself. Then, if there is a need to store the result into a memory location, it also generates that store.

In order to make the needed decisions, we require a data structure that tells us what program variables currently have their value in a register, and which register or registers, if so. We also need to know whether the memory location for a given variable currently has the proper value for that variable, since a new value for the variable may have been computed in a register and not yet stored. The desired data structure has the following descriptors:

1. For each available register, a *register descriptor* keeps track of the variable names whose current value is in that register. Since we shall use only those registers that are available for local use within a basic block, we assume that initially, all register descriptors are empty. As the code generation progresses, each register will hold the value of zero or more names.

2. For each program variable, an *address descriptor* keeps track of the location or locations where the current value of that variable can be found. The location might be a register, a memory address, a stack location, or some set of more than one of these. The information can be stored in the symbol-table entry for that variable name.
```

2019

Group 1

1. (a) What are the different types of languages according to Chomskey's hierarchy? Briefly discuss all of them. Which of them are useful for compiler construction and why? 10

Repeated

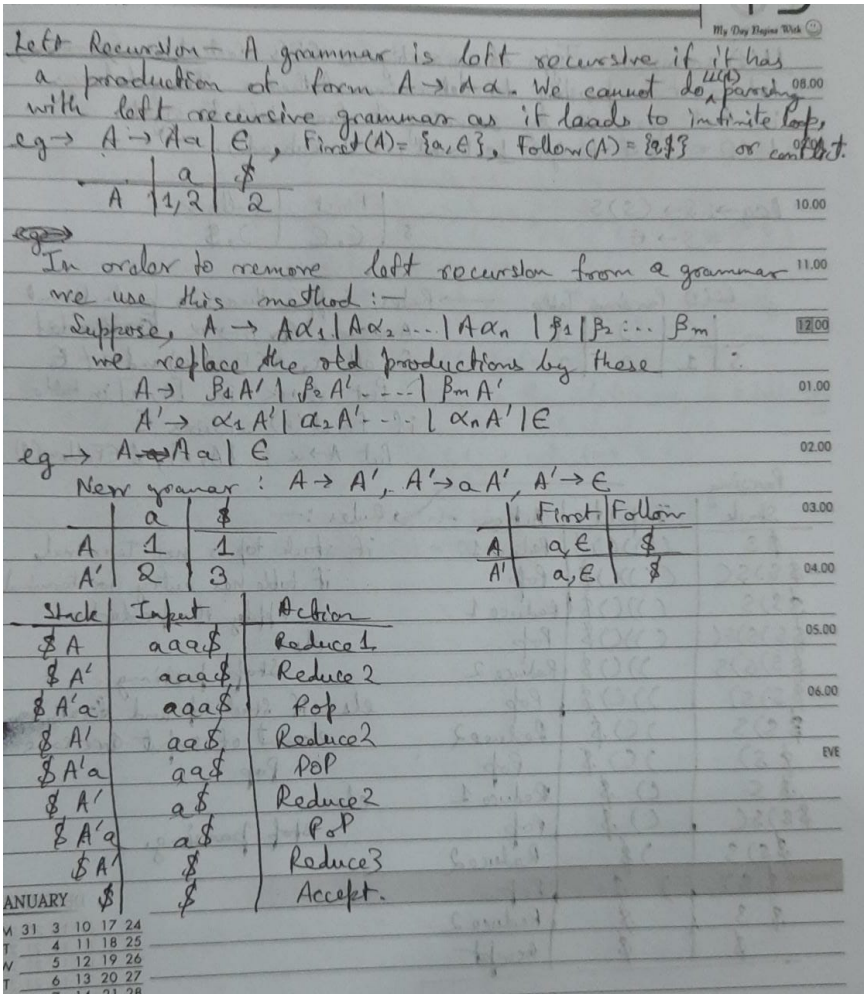
Group 2

4. (a) While constructing a parsing table, why do you need to construct the FIRST and FOLLOW sets? Explain your answer with examples.

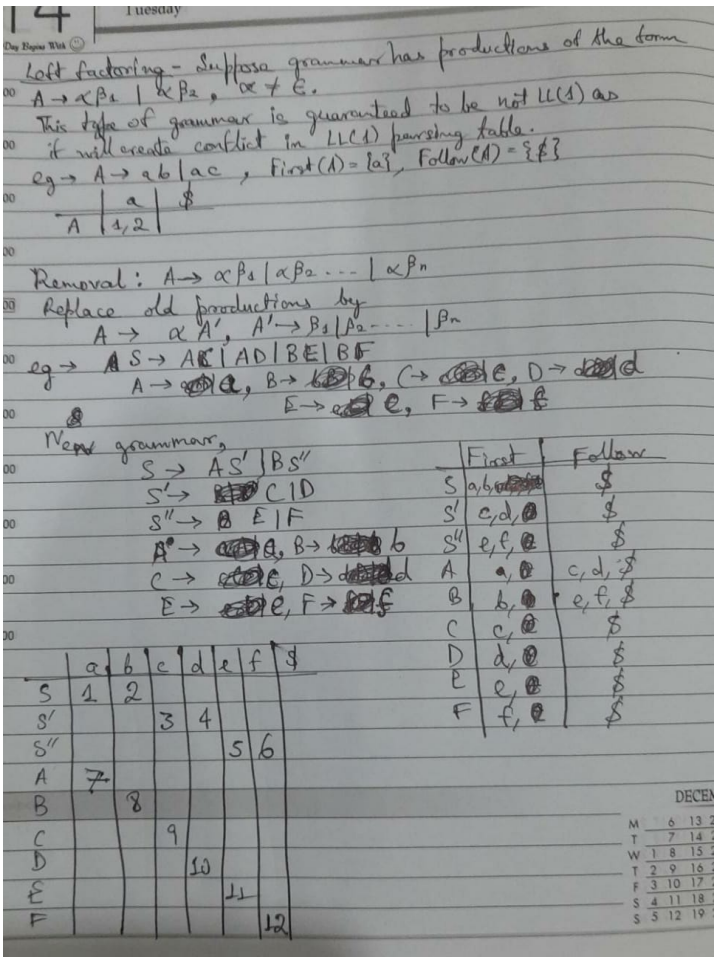
Repeated !

5. (a) With an example explain how do you remove indirect left recursion in a grammar. You may also use the algorithm to explain your answer, (however the algorithm is not necessary).

Repeated !



(c) When is left factoring important in top-down parsing?



6. (a) What are the *closure* and *goto* functions in LR parsing? (No definition is required, explain with an example).

DIY !

Group 3

7. (a) What is *syntax-directed definition (SDD)*? What are synthesized and inherited attributes?
(b) Write an SDD for declaration of list of variables (consider the types 'integer', 'real' and 'char'. Using the SDD, give an annotated parse tree for a declaration statement of two variables of type 'real'. Also draw a dependency graph.
(c) What is an L-attributed definition? 5+(3+2+2)+3=15

Repeated !

8. (a) How do you evaluate synthesized attributes using bottom-up parsers?
(b) What are translation schemes?
(c) Using examples explain how do you evaluate translation schemes for L-attributes definitions? (Use markers for this purpose). 4+3+8=15

Check slides !

9. (a) What is the use of symbol table? In which phases of compiler a symbol table is used?
(b) What are the operations performed on a symbol table when it is implemented as a single hash table?
(c) What is a three-address code? What is 'static single assignment'? How does it differ from three-address code?
(d) Give an example of a 'control-flow graph'.

Group 4

10. (a) What are the main tasks in 'code generation'?
(b) With an example explain what are 'liveness' and 'next use' of variables. Why do you need this information?
(c) Discuss the use of 'Register Descriptor' and 'Address Descriptor'. 2+5+3=10

10b & 10c - repeat

11

(b) With examples discuss the difference between 'useless code (dead code) elimination and unreachable code elimination.
(c) What are the different scopes of code optimization?

Check slides !