SYLLABUS
COMPILER DESIGN
(CIC-303)

Applicable from Batch Admitted in Academic Session 2021-22 Onwards

UNIT I

Compilers and translators, need of translators, structure of compiler: Its different phases, compiler construction tools, Lexical analysis: Role of lexical analyzer, Input Buffering, A simple approach to the design of Lexical Analyzers, Specification and recognition of tokens, Finite automata, From regular expressions to automata, and vice versa, minimizing number of states of DFA, A language for specifying Lexical Analyzers, Design and implementation of lexical analyzer.

UNIT II

The role of the parser, Context free grammars, Writing a grammar: Lexical versus Syntactic analysis, Eliminating ambiguity, Elimination of left recursion, Left factoring, Top Down Parsing: Recursive- Decent parsing, Non-recursive Predictive parsing, LL(1) grammars, Bottom Up Parsing: Shift Reduce Parsing, Operator precedence parsing, LR Parsing: SLR, LALR and Canonical LR parser, Parser Generators.

UNIT III

Syntax Directed Translation: Syntax directed definitions, Evaluation orders for SDD's, construction of syntax trees, syntax directed translation schemes, implementation of syntax directed translation,

Intermediate Code Generation: Kinds of intermediate code: Postfix notation, Parse trees and syntax trees, Three-address code, quadruples and triples, Semantic Analysis: Types and Declarations, Translation of Expressions, Type checking.

UNIT IV

Symbol Table: Symbol tables, its contents, Data Structure for Symbol Table: lists, trees, linked lists, hash tables, Error Detection and Recovery: Errors, lexical phase errors, syntactic phase errors, semantic errors, Error seen by each phase.

Code Optimization: The principal sources of optimizations, Loop optimization, Basic blocks and Flow Graphs, DAG representation of basic blocks, Code Generation: Issues in the design of code generation, A simple target machine mode, A Simple Code Generator, Peep-hole optimization, Register allocation and assignment.

COMPILER DESIGN (ETCS-302)

Instructions to Paper Setters: **MM: 75**

1. Question No. 1 should be compulsory and cover the entire syllabus. Thus question should have objective or short answer type questions. It should be 25 marks.
2. Apart from Question No. 1, rest of the paper shall consists of four units as per the syllabus. Every unit should have two question. However student may be asked to attempt only 1 question from each unit. Each question should be of 12.5 marks.

Objective: This course aims to teach students the principles involved in compiler design. It will cover all the basic components of a compiler, its optimizations and machine code generation. Student will be able to design different types of compiler tools to meet the requirements of the realistic constraints of compilers.

UNIT-I

Brief overview of the compilation process, structure of compiler & its different phases, lexical analyzer, cross compiler, Bootstrapping, quick & dirty compiler. Shift-reduce parsing, operator- precedence parsing, top-down parsing, predictive parsing, LL(1) and LL(k) grammar, bottom up parsing, SLR, LR(1), LALR parsing techniques.

[T1][T2][R1]No. of Hrs. 12]

UNIT-II

Design and implementation of a lexical analyzer and parsing using automated compiler construction tools eg. Lex, YACC, PLY). Syntax-directed translation schemes, implementation of syntax directed translations, intermediate code, postfix notation, three address code, quadruples, and triples, translation of assignment statements. Boolean expressions, control statements, Semantic Analysis, Type Systems, Type Expressions, Type Checker, Type Conversion.

[T2]][R1][R3][R4][R5][No. of Hrs. 12]

UNIT-III

Symbol table, data structures **and implementation of symbol tables. representing** scope information. Run Time Storage **Administration, implementation of a simple stack** allocation scheme, storage allocation in **block structured languages and non block structured languages**. Lexical-phase errors, **sytatic-phase error semantic errors.**

[T1][T2][R2] [No. of Hrs. 10]

UNIT-IV

The principle sources of **optimization, loop optimization, the DAG representation of basic** blocks, value number and **algebraic laws, global dataflow analysis. Object programs**, problems in code generation, **a machine model, a single code generator, register allocation** and assignment, code generation **from DAGs, peephole optimization.**

[T1][T2] [No. of Hrs. 10

COMPLIER DESIGN [CIC-303]

New Topics Added from Academic Session 2021-22 onwards

UNIT - I

Q. 1. What is input buffering? Explain its working in detail?

Ans. The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(**bp**) and forward ptr(**fp**) to keep track of the pointer of the input scanned.

Input buffering is an important concept in compiler design that refers to the way in which the compiler reads input from the source code. In many cases, the compiler reads input one character at a time, which can be a slow and inefficient process. Input buffering is a technique that allows the compiler to read input in larger chunks, which can improve performance and reduce overhead.

1. The basic idea behind input buffering is to read a block of input from the source code into a buffer, and then process that buffer before reading the next block. The size of the buffer can vary depending on the specific needs of the compiler and the characteristics of the source code being compiled. For example, a compiler for a high-level programming language may use a larger buffer than a compiler for a low-level language, since high-level languages tend to have longer lines of code.

2. One of the main advantages of input buffering is that it can reduce the number of system calls required to read input from the source code. Since each system call carries some overhead, reducing the number of calls can improve performance. Additionally, input buffering can simplify the design of the compiler by reducing the amount of code required to manage input.

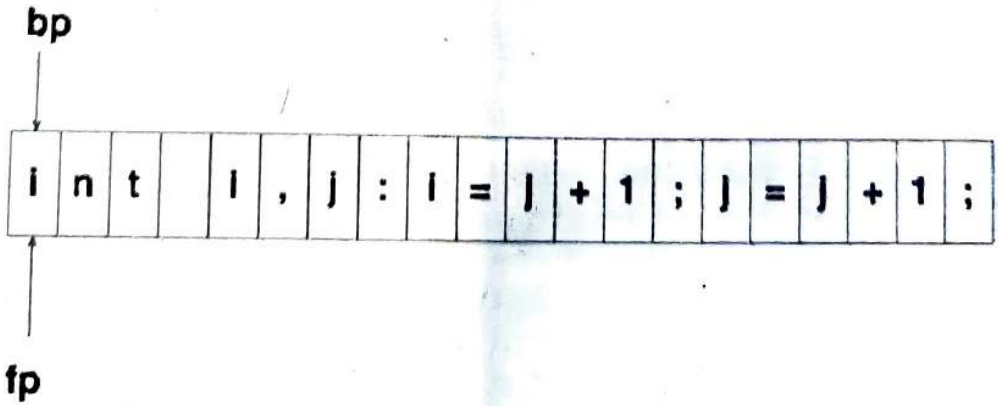
Disadvantages:

For example, if the size of the buffer is too large, it may consume too much memory, leading to slower performance or even crashes. Additionally, if the buffer is not properly managed, it can lead to errors in the output of the compiler.

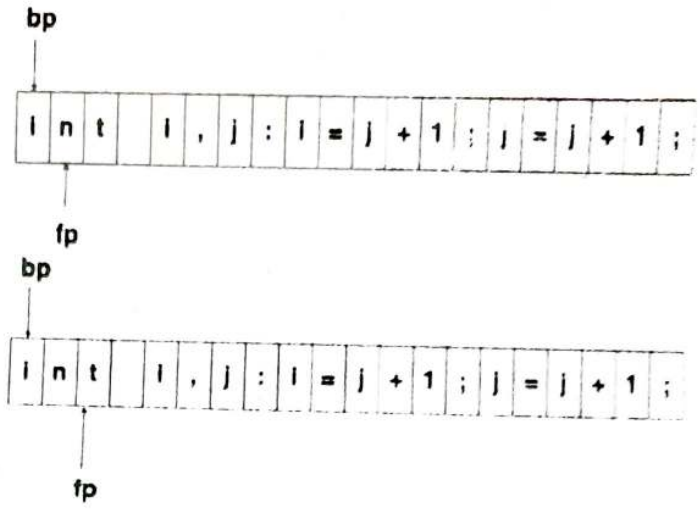
Adantages:

Overall, input buffering is an important technique in compiler design that can help improve performance and reduce overhead. However, it must be used carefully and appropriately to avoid potential problems.

Initially both the pointers point to the first character of the input string as shown below

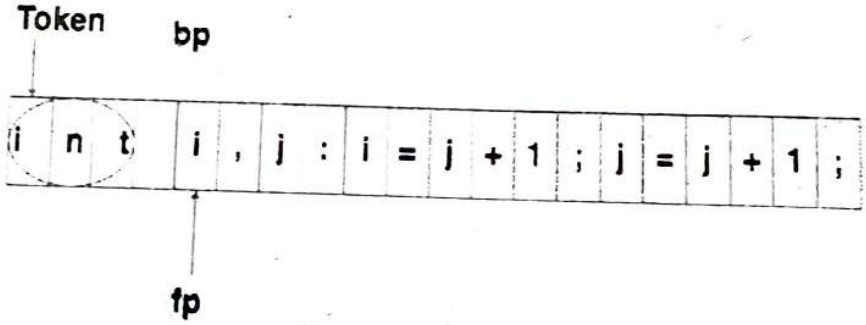


Initial Configuration



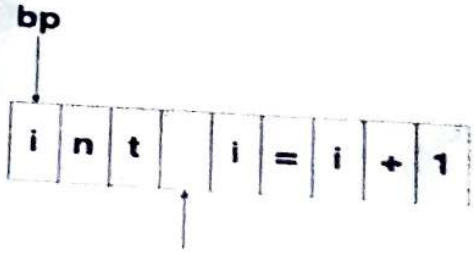
Input Buffering

The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (fp) encounters a blank space the lexeme "int" is identified. The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token. The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used.A block of data is first read into a buffer, and then second by lexical analyzer. there are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme. These are explained as



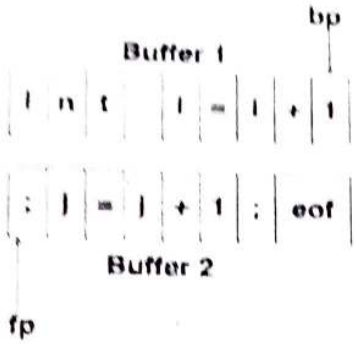
Input buffering

1. One Buffer Scheme: In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.



One buffer scheme storing input string

2. Two Buffer Scheme: To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. The first buffer and second buffer are scanned alternately. When end of current buffer is reached the other buffer is filled. The only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely. Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves towards right in search of end of lexeme. As soon as blank character is recognized, the string between bp and fp is identified as corresponding token. To identify, the boundary of first buffer end of buffer character should be placed at the end first buffer. Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. When fp encounters first **eof**, then one can recognize end of first buffer and hence filling up second buffer is started. In the same way when second **eof** is obtained then it indicates of second buffer. Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified. This **eof** character introduced at the end is calling **Sentinel** which is used to identify the end of buffer.



Two buffer scheme storing input string

Advantages:

Input buffering can reduce the number of system calls required to read input from the source code, which can improve performance.

Input buffering can simplify the design of the compiler by reducing the amount of code required to manage input.

Disadvantages:

If the size of the buffer is too large, it may consume too much memory, leading to slower performance or even crashes.

If the buffer is not properly managed, it can lead to errors in the output of the compiler.

Overall, the advantages of input buffering generally outweigh the disadvantages when used appropriately, as it can improve performance and simplify the compiler design.

Q.2. Explain lexical analysis phase in compiler.

Ans. Lexical analysis is the first phase of a compiler. It takes modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

There are three terminologies

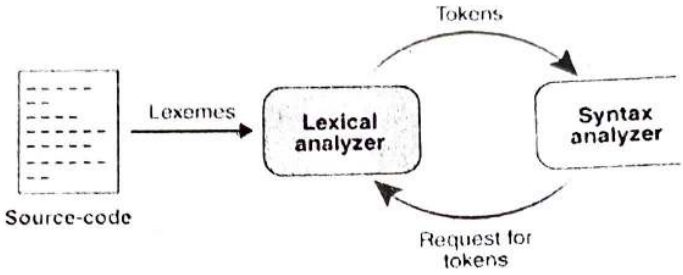
(a) Token: It is a sequence of characters that represents a unit of information in the source code

(b) Pattern: The description used by the token is known as a pattern.

(c) Lexeme: A sequence of characters in the source code, as per the matching pattern of a token, is known as lexeme. It is also called the instance of a token.

The Architecture of Lexical Analyzer

To read the input character in the source code and produce a token is the most important task of a lexical analyzer. The lexical analyzer goes through with the entire source code and identifies each token one by one. The scanner is responsible to produce tokens when it is requested by the parser. The lexical analyzer avoids the whitespace and comments while creating these tokens. If any error occurs, the analyzer correlates these errors with the source file and line number.



Roles and Responsibility of Lexical Analyzer

The lexical analyzer performs the following tasks-

- The lexical analyzer is responsible for removing the white spaces and comments from the source program.
- It corresponds to the error messages with the source program.
- It helps to identify the tokens.
- The input characters are read by the lexical analyzer from the source code.

Advantages of Lexical Analysis

- Lexical analysis helps the browsers to format and display a web page with the help of parsed data.
- It is responsible to create a compiled binary executable code.
- It helps to create a more efficient and specialised processor for the task.

Disadvantages of Lexical Analysis

- It requires additional runtime overhead to generate the lexer table and construct the tokens.
- It requires much effort to debug and develop the lexer and its token description.
- Much significant time is required to read the source code and partition it into tokens.

Q.3. Explain finite automata ?

Ans. Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly. Finite automata is a recognizer for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand.

The mathematical model of finite automata consists of:

- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One Start state (q0)
- Set of final states (qf)
- Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), $Q \times \Sigma \rightarrow Q$

Finite Automata Construction

Let L(r) be a regular language recognized by some finite automata (FA).

- **States :** States of FA are represented by circles. State names are written inside circles.

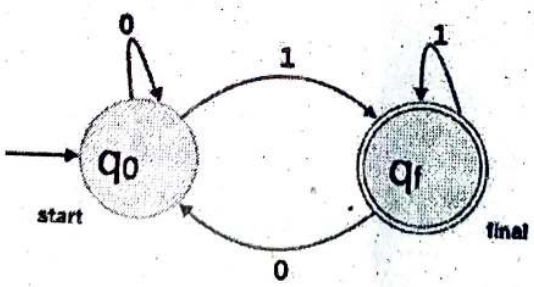
- **Start state :** The state from where the automata starts, is known as the start state. Start state has an arrow pointed towards it.

- **Intermediate states :** All intermediate states have at least two arrows; one pointing to and another pointing out from them.

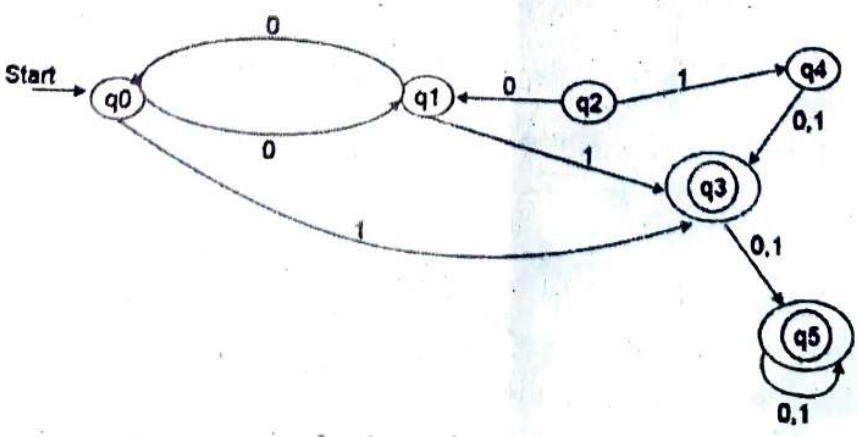
- **Final state :** If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles. It may have any odd number of arrows pointing to it and even number of arrows pointing out from it. The number of odd arrows are one greater than even, i.e. odd = even+1.

- **Transition :** The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows points to the destination state. If automata stays on the same state, an arrow pointing from a state to itself is drawn.

Example : We assume FA accepts any three digit binary value ending in digit 1.
FA = {Q(q0, qf), Σ(0,1), q0, qf, δ}



Q. 4. Minimise the number of states in the following diagram:



Ans. Minimization of DFA means reducing the number of states from given FA. Thus, we get the FSM (finite state machine) with redundant states after minimizing the FSM. We have to follow the various steps to minimize the DFA. These are as follows:

Rules:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

Step 4: Find similar rows from T1 such that:

1. $\delta(q, a) = p$
2. $\delta(r, a) = p$

That means, find the two states which have the same value of a and b and remove one of them.

Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

Solution:

Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.

Step 2: Draw the transition table for the rest of the states.

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q5	q5
*q5	q5	q5

Step 3: Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final states:

State	0	1
q0	q1	q3
q1	q0	q3

2. Another set contains those rows, which starts from final states.

State	0	1
q3	q5	q5
q5	q5	q5

Step 4: Set 1 has no similar rows so set 1 will be the same.

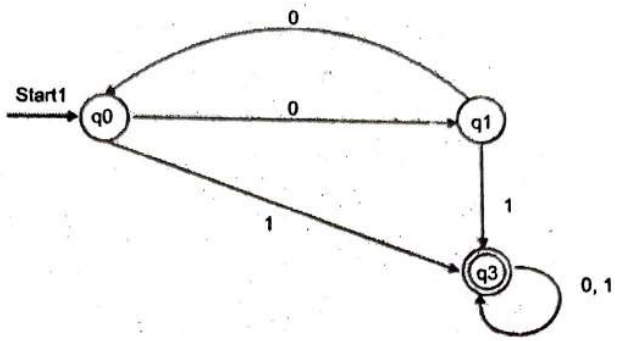
Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to the same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

State	0	1
q3	q3	q3

Step 6: Now combine set 1 and set 2 as:

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q3	q3

Now it is the transition table of minimized DFA.



Q. 5. Convert Regular expression into Finite automata. Design a FA from given regular expression 10 + (0 + 11)0* 1.

Ans. To convert the RE to FA, we are going to use a method called the subset method. This method is used to obtain FA from the given regular expression. This method is given below:

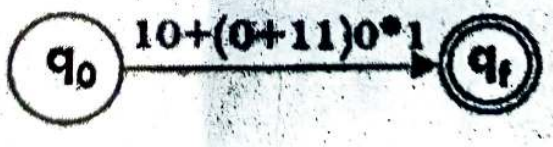
Step 1: Design a transition diagram for given regular expression, using NFA with ε moves.

Step 2: Convert this NFA with ε to NFA without ε.

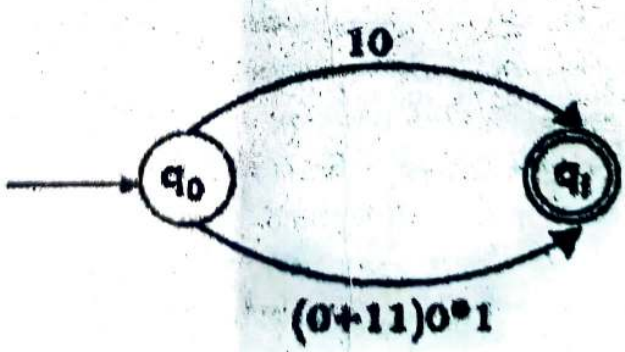
Step 3: Convert the obtained NFA to equivalent DFA.

First we will construct the transition diagram for a given regular expression.

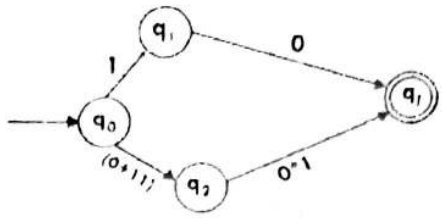
Step 1:



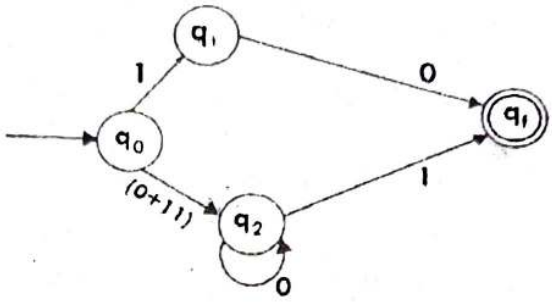
Step 2:



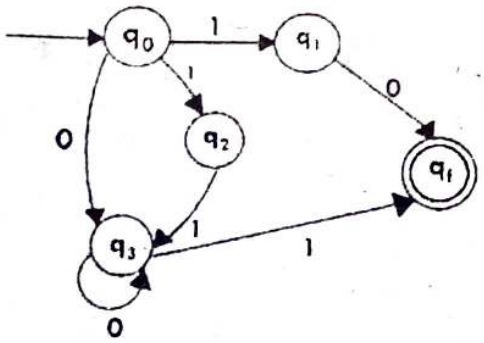
Step 3:



Step 4:



Step 5:



Now we have got NFA without ε. Now we will convert it into required DFA for that, we will first write a transition table for this NFA.

tate	0	1
→q0	q3	{q1, q2}
q1	qf	ϕ
q2	ϕ	q3
q3	q3	qf
*qf	ϕ	ϕ

The equivalent DFA will be:

State	0	1
→[q0]	[q3]	[q1, q2]
[q1]	[qf]	ϕ
[q2]	ϕ	[q3]
[q3]	[q3]	[qf]
[q1, q2]	[qf]	[qf]
*[qf]	ϕ	ϕ

END TERM EXAMINATION [JULY 2023]
SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS-302]

Time: 3 Hrs. Max. Marks: 75

Note: Attempt five questions in all including Q. No. 1 which is compulsory. Select one question from each unit.

Q.1. Answer the following questions: (5 × 5 = 2.5)

Q.1. (a) What is Backpatching? Explain with the help of example.

Q.1. (b) Distinguish between Compiler and Interpreter briefly?

Q.1. (c) Explain various principle sources of Code Optimization.

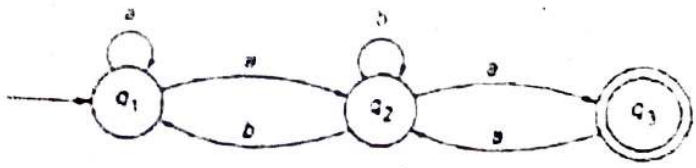
Q.1. (d) What is DAG? What are the advantages of DAG?

Q.1. (e) What is the role and various functions of Lexical analyser? Explain briefly.

UNIT - I

Q.2. (a) Discuss the architecture of Compiler with its phases in details (7.5)

Q.2. (b) Consider the transition diagram. (5)



Convert the above finite automata into the regular expression.

Q.3. (a) What is bottom-up parsing? How to construct the SLR parsing table? Explain with the help of an example. (6.5)

Q.3. (b) For the Grammar given below: (6)

E-> TE'

E'-> + TE'/ε

T-> FT'

F-> *FT'//ε

F-> (E)/id

Construct the Predictive Parsing Table. Whether this grammar is LL(1) or not?

UNIT - II

Q.4. (a) What is syntax directed translation? How the syntax directed translation schemes are described. Explain with the help of example. (6)

Q.4. (b) What is intermediate code? What are the various ways to represent the intermediate code? Explain postfix notation, parse trees and syntax trees, three address code, quadruples, and triples with the help of examples. (6.5)

Q.5. (a) Describe the language for specifying Lexical Analyser with the help of example. (6)

Q.5. (b) Write a short note on: Type Checker, Type Conversion and Boolean Expressions. (6.5)

Q.6. (a) What information is contained by the Symbol Table? Explain the contents and its capabilities of symbol table. (7)

Q.6. (b) What is error? Explain sources of errors and explain various types of errors at each phase of compiler. (5.5)

Q.7. (a) Describe various data structures of symbol tables in detail. (5.5)

Q.7. (b) What are the different storage allocation strategies in the runtime environment of the compiler? Explain. (7)

UNIT - IV

Q.8. (a) What is code generation? Explain various issues of Code Generation. (6.5)

Q.8. (b) Briefly eplains.

(a) Code generation from DAGs

(b) Value number and algebraic laws. (6)

Q.9. (a) How Peep Hole Optimization is useful in Code Optimization & Code Generation phase. Explain briefly. (5.5)

Q.9. (b) What are basic blocks and Flow Graphs? Explain with the help of examples. (7)

FIRST TERM EXAMINATION [FEB. 2016]
SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS–302]

Time : 1.5 hrs. **M.M. : 30**

Note: *Attempt any three questions including Q. no. 1 which is compulsory.*

Q.1. (a) Discuss Merits and Demerits of Single pass compiler and Multi pass compiler.

Ans: Single pass Compiler is a compiler that passes through the parts of each compilation unit only once, immediately translating each part into its final machine code.

Merits:

1. They are faster.
2. It is also called as a Narrow compiler.
3. The external storage for the intermediate file between two passes is slow or is inconvenient to use.

Demerits:

1. It has less efficient code optimization and code generation.
2. Large Memory is required by compiler.
3. It has limited scope of passes.

Multipass Compiler: It reads the program several times. Each time transforming it in to different form.

Merits:

1. Better code optimization and code generation.
2. It is called wide compiler as they scan each and every portion of program.

Demerits:

1. Slower, as no. of passes means more execution time.

Q. 1. (b) What do you mean by Bootstrapping in Compilation?

Ans: Bootstrapping is the process of writing a compiler (or assembler) in the source programming language that it intends to compile. Applying this technique leads to a self-hosting compiler. An initial minimal core version of the compiler is generated in a different language; from that point, successive expanded versions of the compiler are run using the minimal core of the language.

Bootstrapping a compiler has the following advantages:

- It is a non-trivial test of the language being compiled, and as such is a form of dogfooding.
- Compiler developers only need to know the language being compiled.
- Compiler development can be done in the higher level language being compiled.
- Improvements to the compiler's back-end improve not only general purpose programs but also the compiler itself.
- It is a comprehensive consistency check as it should be able to reproduce its own object code.

1.Q. (c) What is the need for separating the Parser from Scanner?

Ans: Lexer: character stream to token stream

Parser: token stream to syntax tree

Advantages:

Simpler design: Based on related but distinct theoretical underpinnings Compartmentalizes some low-level issues, e.g., I/O, internationalization.

Faster

Lexing is time-consuming in many compilers (40-60% ?)

By restricting the job of the lexer, a faster implementation is usually feasible.

Q.1. (*d*) Find left most derivation, right most derivation, and derivation tree of a string baababa.

{S → bB | aA, A→b | bS | aAA, B→ a | aS | bBB

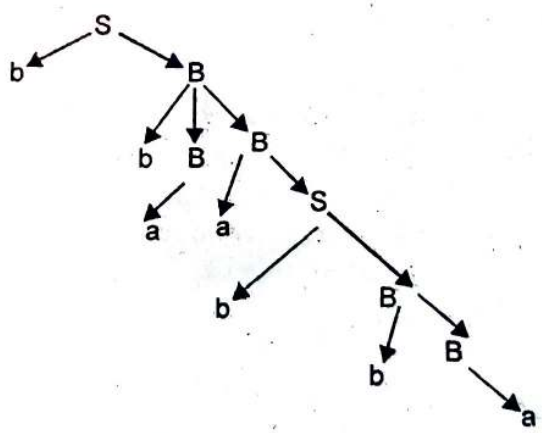
Ans: Left Most Derivation:

S –> bB
S –> bbBB
S –> bbaB
S –> bbaaS
S –> bbaabB
S –> bbaabaS
S –> bbaabab B
S –> bb aababa

Right Most Derivation

S –> bB
S –> bbBB
S –> bbBaS
S –> bbBabS
S –> bbBabaS
S –> bbBababB
S –> bbBababa
S –> bbaababa

Derivation tree



Q.1. (*e*) Explain handle pruning with an example.

Ans: A right most derivation in reverse can be obtained by *handle pruning*.

i.e., start with a string of terminals w that is to parse. If w is a sentence of the grammar at hand, then $w = \gamma_n$, where γ_n is the nth right sentential form of some as yet unknown rightmost derivations $\gamma_{n-1}\ \gamma_n = w$.

Example for right sentential form and handle for grammar

- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow (E)$
- $E \rightarrow id$

Right sentential form	Handle	Reduction production
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Q.1. (f) What are the responsibilities of Loader, Linker, and Assembler in the compiler environment?

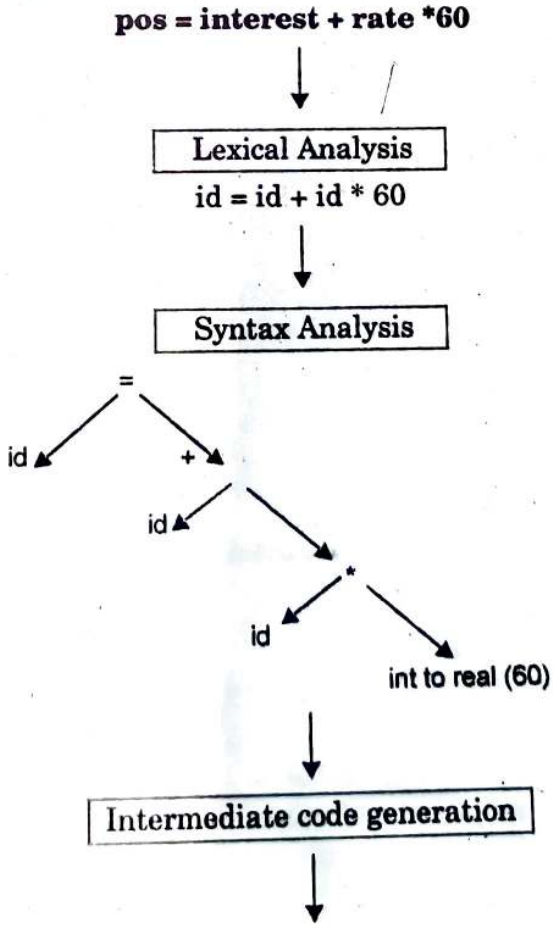
Ans: Assembler: An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

Linker: Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/ routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

Loader: Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

Q.1. (g) Write steps involved in compilation of following sentence pos = interest + rate * 60.

Ans:



T1 = in to real (20)
T2 = id * T1
T3 = id + T2
id = T3

↓

Code optimization

↓

T2 = id * T1
T3 = id + T2
id = T3

↓

Code generation

↓

Assembly code

Q2. Attempt both parts

(a) Remove left recursion from the following grammar:

$$S \rightarrow Aa/b$$
$$A \rightarrow Ac/Sd/e$$

Ans: As we don't have immediate left recursion,

Step (1) Rename S, A as A_1, A_2 respectively

$$A_1 \text{ -> } A_2a \mid b$$
$$A_2 \text{ -> } A_2\ c \mid A_1\ d \mid e$$

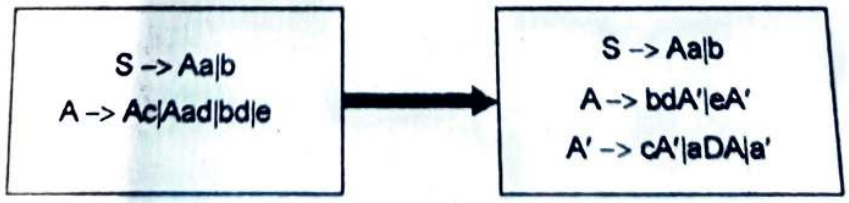
Step (2) Substitute value A_1 of statement (1) in R.H.S of statement (2)

$$A_1 \text{ -> } A_2a \mid b$$
$$A_2 \text{ -> } A_2\ c \mid A_2\ a\ d \mid bd \mid e$$

Step (3) Again substitute $A_1 = S, A_2 = A$

$$S \text{ -> } Aa \mid b$$
$$A \text{ -> } Ac \mid Aad \mid bd \mid e$$

Step (4) It has now **immediate left recursion**



Q.2. (b) Prove whether following:

$$S \longrightarrow XS \mid dS \mid \in$$
$$X \longrightarrow Y \mid Zb \mid aY$$
$$Y \longrightarrow cZ$$
$$Z \longrightarrow e$$

Ans: A Grammar is LL(1) if for every production A $\alpha \mid \beta \longrightarrow$

First (α) ^ First (β) = ϕ

If $\lambda \in$ First (α) then First (β) ^ Follow (A) = ϕ

If $\lambda \in$ First (β) then First (α) ^ Follow (A) = ϕ

First(S) = First(X) –> First(Y) –> First(Z) –> e

First(X) = First(Y) –> First(Z) –> e

First(Y) = First(Z) –> e

First(Z) = e

First (S) = {e} $\cup$ {d} $\cup$ {ϵ}

First (X) = {c} $\cup$ {e} $\cup$ {a}

This Grammar is LL(1)

Q.3. Write Algorithm to construct SLR parser. Show all the steps of SLR parsing for following grammar.

$$S \rightarrow R$$
$$R \rightarrow Rb$$
$$R \rightarrow a$$

Ans: Simple LR or SLR parser is a type of LR parser with small parse tables and a relatively simple parser generator algorithm. As with other types of LR(1) parser, an SLR parser is quite efficient at finding the single correct bottom-up parse in a single left-to-right scan over the input stream, or backtracking. The parser is mechanically generated from a formal grammar for the language.

Algorithm:

Input: An Augmented Grammar G'

Output: goto

Method:

1. Initially construct set of items

C = {I_0, I_1, I_2, ... In} where C is a collection of LR(0) items for grammar

2. Parsing actions are based on each item or state I_i.

Various Actions are:

(*a*) If $\rightarrow \alpha.a\beta$ is in I_i and goto (I_i,*a*) = I_j then set Action [*i*, a] = "shift j".

(*b*) If $A \rightarrow \alpha.$ is in I_i then set Action[i , a] to reduce A –> α for all symbols a, where a belongs to Follow(A).

(*c*) If $S \rightarrow S.$ is in I_i then entry in action table Action [*i* , $] = "accept"

3. The go to part of SLR table can be filled as: The go to transitions for state *i* is considered for non termialsonly. If goto(I_i, *a*) = I_j then goto [*i* ,*A*] = *j*.

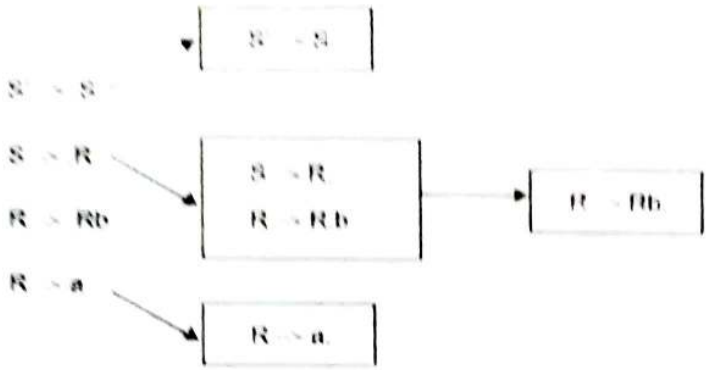
4. All entries not defined by rule 2 and 3 are considered to be "error".

S R →

RRb →

Ra →

Item set 10



Q.4. Attempt both parts

(a) Draw the finite automata which can be used by lexical analyser to recognise identifier, number and operators.

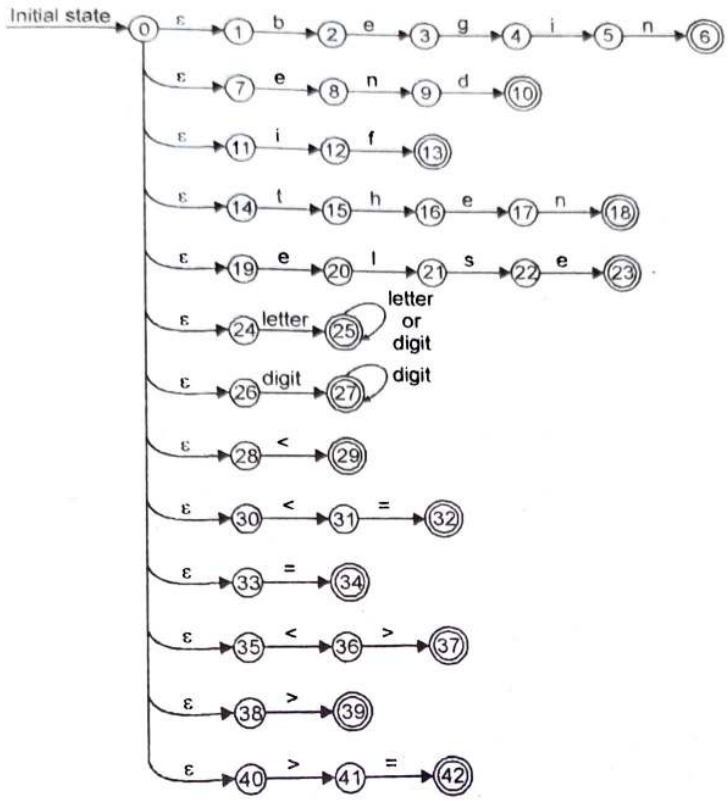
Ans: Auxiliary definations

letter = A|B|C|.....|Z

digit = 0|1|2|.....|9

Translation rules

begin	{return 1}
end	{return 2}
if	{return 3}
then	{return 4}
else	{return 5}
letter (letter + digit)*	{Value = install ( ): return 6}
digit+	{Value = install ( ): return 7}
<	{Value = 1 : return 8}
<=	{Value = 2 : return 8}
=	{Value = 3 : return 8}
<>	{Value = 4 : return 8}
>	{Value = 5 : return 8}
>=	{Value = 6 : return 8}



Q.4. (*b*) What is operator grammar? Write operator precedence parsing table for following grammar:

$$E \rightarrow E + E|E*E|(E)|a$$

Ans: An operator precedence grammar is a context-free grammar that has the property that no production has either an empty right-hand side or two adjacent non terminals in its right-hand side. These properties allow precedence relations to be defined between the terminals of the grammar. A parser that exploits these relations is considerably simpler than more general-purpose parsers such as LALR parsers. Operator-precedence parsers can be constructed for a large class of context-free grammars.

First attach $ at starting and ending of string i.e. $ a1 + a2 + a3$.

Put precedence relation between operators & symbos using precedence relation table

$\$ \rightarrow \cdot a1 <\cdot + <\cdot a2 \cdot> * <\cdot a3 \cdot> \$$

String	Handle	Production Used
$<·a1·> + <·a2·> *<·a3·>$	<·a1·>	E -> a
$E + <·a2·> *<·a3·>$	<·a2·>	E -> a
$E + E *<·a3·>$	<·a3·>	E -> a
$E + E * E$	Remove all non terminals	
$+*$	Insert Precedence relation between operators	
$<·+<·*·>$	<·*·> i.e. E*E	E -> E * E
$<· +·>$	<·+·> i.e. E+E	E -> E + E
$		

ND TERM EXAMINATION [APRIL 2016]
SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS–302]

Time : 1 A Hrs.

M.M. : 30

Note : Attempt any three questions including Q. no. 1 which is compulsory.

Q 1. Attempt any four parts

(a) What are the different representations of three address statements?

Ans. Three-Address Code: Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example

```
a = b + c * d;
```

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;
r2 = b + r1;
r3 = r2 + r1;
a = r3
```

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format.

Op	arg_1	arg_2	Result
*	C	d	r1
+	B	r1	r2
+	r2	r1	r3
=	r3		A

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2. The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG

Op	arg_1	arg_2
*	c	D
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Declarations

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

Example:

We take the example of C programming language where an integer variable is assigned 2 bytes of memory and a float variable is assigned 4 bytes of memory.

int a;
float b;
Allocation process:
{offset = 0}
int a;
id.type = int
id.width = 2
offset = offset + id.width
{offset = 2}
float b;
id.type = float
id.width = 4
offset = offset + id.width
{offset = 6}

To enter this detail in a symbol table, a procedure *enter* can be used. This method may have the following structure:

enter(name, type, offset)

This procedure should create an entry in the symbol table, for variable *name*, having its type set to type and relative address *offset* in its data area.

Q.1. (*b*) Give three address code for the following code segment:

While(a<b) do

If(c<d) then x = y+ z

Ans: 1. If a<b go to 3
2. goto(8)
3. If c<d goto (5)
4. goto(8)
5. t1 = y+z
6. x= t1
7. goto t1
8. end

Q.1 (*c*) Write Syntax directed Translation scheme for While Loop.

Ans: Syntax directed Translation scheme for While Loop:

- Consider the production S → while (C) S1 for while-statements. The shape of the code for implementing this production can take the form:
  label L1: // beginning of code for S
  code to evaluate C
  if C is true gotoC.true // C.true will be set to L2
  if C is false gotoC.false // C.false will be set to S.next
  label L2:
  code to evaluate S1
  goto L1
  S.next: // this is where control flow will go after executing S
- Here is an SDD for this translation:

```
S → while ( C ) S1  {
        L1 = newlabel();
        L2 = newlabel();
        S1.next = L1;
    C.false = S.next;
    C.true = L2;
    S.code = label || L1 || C.code || label || L2 || S1.code
    }
```

Q.1. (*d*) Differentiate between SDT and SDD

Ans: SDT: It is a kind of notation in which each production of CFG is associated with a set of semantic rules or actions and each grammar symbol is associated with a set of attributes. Therefore the grammar and the semantic actions combine to make a Syntax Directed Translation.

It is of two types:

1. **Synthesized Translation:** These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

$S \rightarrow ABC$

If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

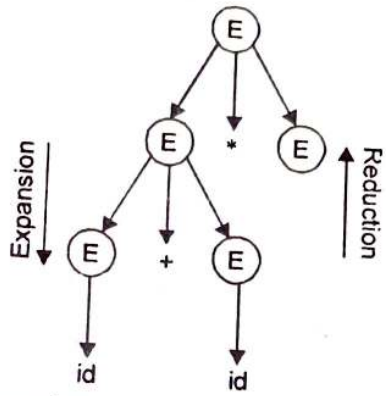
As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

2. **Inherited Translation:** In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production.

S → ABC

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Expansion: When a non-terminal is expanded to terminals as per a grammatical rule



Reduction : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks. Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis). Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

For example:

int value = 5;

<type, "integer">

<presentvalue, "5">

For every production, we attach a semantic rule.

SDD (Syntax Directed Defination): A CFG where each grammar production A –> α is associated with a set of semantic rules of the form b = f(c1, c2, ..., ck); where: b is a synthesized attribute of A or an inherited attribute of one of the grammar symbols in α c1, c2, ... are attributes of the symbols used in the production.

Synthesized and Inherited Attributes

- Attributes are values computed at the nodes of a parse tree.
- *Synthesized attributes* are values that are computed at a node *N* in a parse tree from attribute values of the children of *N* and perhaps *N* itself. The identifiers $$, $1, $2, etc., in Yacc actions are synthesized attributes. Synthesized attributes can be easily computed by a shift-reduce parser that keeps the values of the attributes on the parsing stack.
- An SDD is *S-attributed* if every attribute is synthesized.
- *Inherited attributes* are values that are computed at a node *N* in a parse tree from attribute values of the parent of *N*, the siblings of *N*, and *N* itself.

- An SDD is *L-attributed* is every attribute is either synthesized or inherited from the left.

Q.1. (c) Write brief note on responsibilities of semantic analysis phase.

Ans: Responsibilities of semantic analysis phase: Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

For example:

int a = "value";

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking.

Q.2. Attempt both parts

(a) What is syntax directed Translation? How are semantic actions attached to the production? Explain with Example.

Ans: SDT: It is a kind of notation in which each production of CFG is associated with a set of semantic rules or actions and each grammar symbol is associated with a set of attributes. Therefore the grammar and the semantic actions combine to make a Syntax Directed Translation.

Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal.

Typically, SDT's are implemented during parsing, without building a parse tree. SDT's can be used to import classes of SDD's:

- The underlying grammar is LR-parsable, and the SDD is S-attributed.
- The underlying grammar is LL-parsable, and the SDD is L-attributed.

In both these cases, the semantic rules in an SDD can be converted into an SDT with actions that are executed at the right time. During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of the action have been matched.

SDT's that can be implemented during parsing can be characterized by introducing distinct *marker nonterminals* in place of each embedded action; each marker M has only one production, $M \rightarrow \varepsilon$. If the grammar with marker non-terminals can be parsed by a given method, then the SDT can be implemented during parsing.

Semantic Action: It is an action which is executed whenever parser will recognise the input string generated by CFG

Ex: A - > BC {Semantic Action}

Semantic Action is written in curly braces attached with a production.

Example: Grammar given is:

S -> E$
E -> E+ E
E -> E*E
E -> (E)
E -> I

I –> I digit
I –> digit where digit = 0|1|..9

SDT fot it:

S.No.	Production	Semantic Action
1.	S – > E$	{Print E.VAL}
2.	E – > $E^{(1)} + E^{(2)}$	{E.VAL = $E^{(1)}$.VAL + $E^{(2)}$.VAL}
3.	E – > $E^{(1)} * E^{(2)}$	{E.VAL = $E^{(1)}$.VAL * $E^{(2)}$.VAL}
4.	E – >($E^{(1)}$)	{E.VAL = $E^{(1)}$.VAL}
5.	E – > I	{E.VAL = I.VAL }
6.	I – > $I^{(1)}$ digit	{I.VAL = 10 * $I^{(1)}$.VAL + LEXVAL}
7.	I – > digit	{$I^{(1)}$.VAL = LEXVAL}

Q.2. (*b*) Generate the intermediate code and output E. True list and E. False list for the following expression:

x>y and z>k and r<s from the following grammar

E - > E or ME

E - > E and ME

E - > not E

E - > ϵ

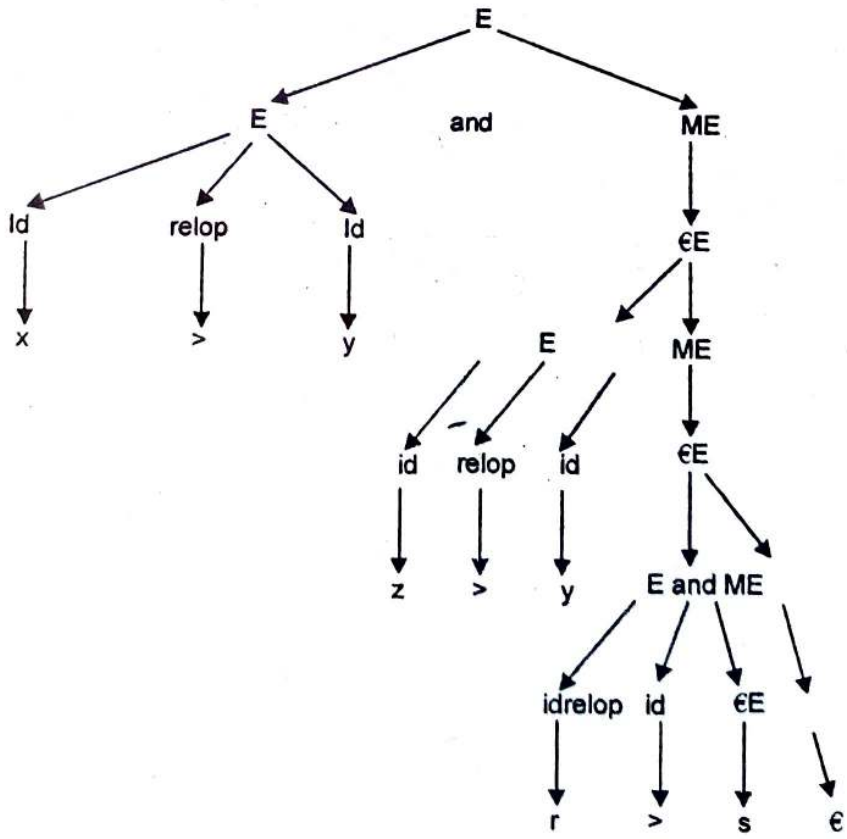
E - > id

E - > id relop id

M - > ϵ

Build up the parse tree.

Ans:.



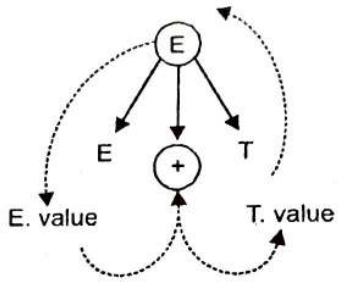
Q.3. Attempt both parts

(a) Explain L-attributed and S-attributed grammer with example.

Ans: S-Attributed Grammars are a class of attribute grammars characterized by having no inherited attributes, but only synthesized attributes. Inherited attributes, which must be passed down from parent nodes to children nodes of the abstract syntax tree during the semantic analysis of the parsing process, are a problem for bottom-up parsing because in bottom-up parsing, the parent nodes of the abstract syntax tree are created *after* creation of all of their children. Attribute evaluation in S-attributed grammars can be incorporated conveniently in both top-down parsing and bottom-up parsing.

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).

E. Value = E. value + T. value



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed grammars are a special type of attribute grammars. They allow the attributes to be evaluated in one depth-first left-to-right traversal of the abstract syntax tree. As a result, attribute evaluation in L-attributed grammars can be incorporated conveniently in top-down parsing.

A syntax-directed definition is L-attributed if each inherited attribute of Xj on the right side of

$$A \rightarrow X1\ X2\ ..Xn$$

depends only on

1. The attributes of the symbols X1, X2, .., Xj-1
2. The inherited attributes of A

Every S-attributed syntax-directed definition is also L-attributed.

Implementing L-attributed definitions in Bottom-Up parsers requires rewriting L-attributed definitions into translation schemes.

Many programming languages are L-attributed. Special types of compilers, the narrow compilers, are based on some form of L-attributed grammar. These are a strict superset of S-attributed grammars. Used for code synthesis.

Either "Inherited attributes." or "synthesized attributes" associated with the occurrence of symbol X1, X2...Xn.

Q.3. (b) Write syntax directed translation scheme for multi-dimensional arrays with suitable example.

Ans: Similar Example:

Consider the following CFG, which defines expressions that use the three operators: +, &&, ==. Let's define a syntax-directed translation that type checks these expressions; i.e., for type-correct expressions, the translation will be the type of the expression (either INT or BOOL), and for expressions that involve type errors, the translation will be the special value ERROR. We'll use the following type rules:

1. Both operands of the + operator must be of type INT.
2. Both operands of the && operator must be of type BOOL.
3. Both operands of the == operator have the same (non-ERROR) type.

Here is the CFG and the translation rules:

CFG	Translation rules
exp ->exp + term	if ((exp_2.trans == INT) and (term.trans == INT) then exp_1.trans = INT else exp_1.trans = ERROR
exp ->exp&& term	if ((exp_2.trans == BOOL) and (term.trans == BOOL) then exp_1.trans = BOOL else exp_1.trans = ERROR
exp ->exp == term	if ((exp_2.trans == term.trans) and (exp_2.trans != ERROR)) then exp_1.trans = BOOL else exp_1.trans = ERROR
exp -> term	exp.trans = term.trans
term -> true	term.trans = BOOL
term -> false	term.trans = BOOL
term ->intliteral	term.trans = INT
term -> (exp)	term.trans = exp.trans

Input

=====

(2 + 2) == 4

Q.4. Attempt both parts:

(*a*) What is lexical phase, syntactic and semantic phase errors?

Ans: Lexical phase errors: Lexical Phase breaks the program in to tokens i.e it recognizes the tokens, in program.

1. If after some processing, the lexical Analyzer discovers that no prefix of the remaining input belongs to any token class, then lexical errors are generated.
2. It is mainly generated because next token to be read in source program is misspelled.

Eg: Integer Constant out of bound(0 to 32767)

Syntactic Phase Errors: These errors occur when stream of token not follow the grammatical rules of programming language.

1. Missing ; at the end of the statement.
2. Missing Right Parenthesis (a +(b+c)
3. Misspelled keyword
4. Colon in place of semicolon ex: I = 1:

Semantic phase errors

1. Non Declared variables
2. Type conflict of operands
3. Incorrect procedure calls (e.g.Wrong no. of Parameters).

Q.4. (*b*) Discuss the various data structures used for symbol table with example.

Ans: Different data structures used for symbol table are:

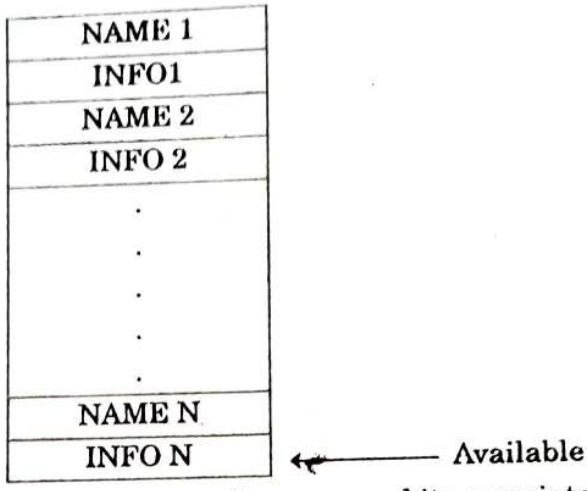
1. Lists
2. Self Organising Lists

3. Trees

4. Hash Tables

LISTS:

1. It is simple and easy to implement data structure for symbol table in linear list records:



2. One can use single array to store the name and its associated information.

3. New names are added to end of list on FCFS basis.

4. Method of access is sequential.

Advantages:

1. Minimum soace requirement.

2. Easy to understand and implement.

Disadvantages:

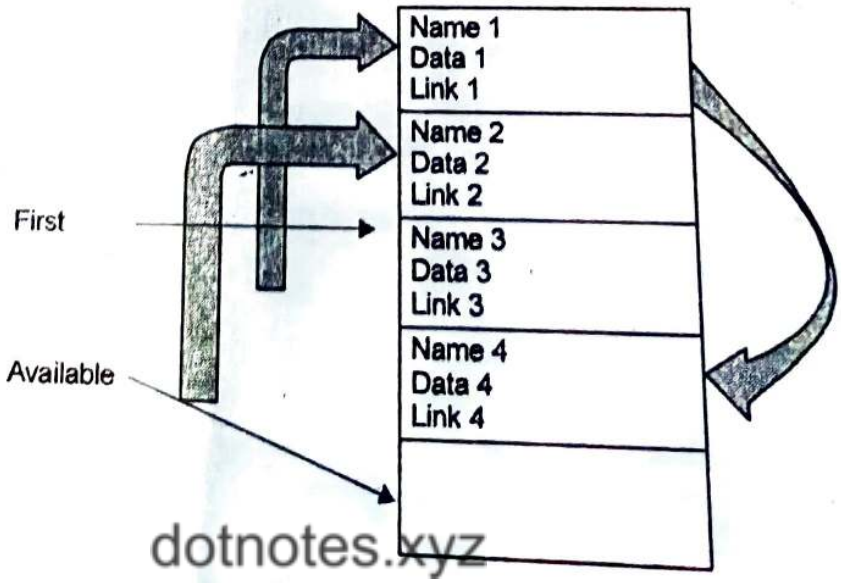
1. Sequential Access.

2. Amount of work done required to be done is high.

SELF ORGANISING LISTS:

1. It is used to decrease searching time in list organisation.

2. Special attribute link is added to information of name that allows dynamic features in list, this feature help in minimizing wastage of space and also promotes reusability to some extent.



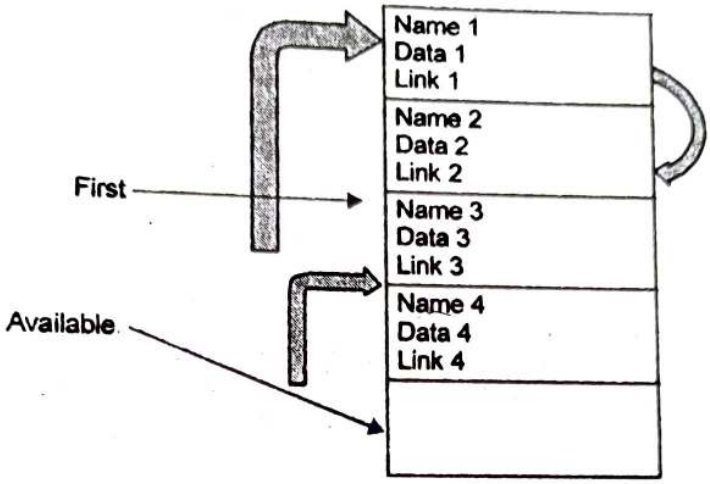
- First pointer points to the beginning of symbol table
- If we have names in the following order

3 - > 1 - > 4- > 2

- If we need a particular record we need to move record to the front of the list.

Suppose we need 4, so new connection will be

4 -> 3 - > 1 - > 2



Mechanism

(*a*) Initial stage 3 - > 1 - > 4 - > 2

(*b*) 4th name is required

(*c*) move 4th to front

3 – > 1 - > 4 - > 2

(*d*) 4 - > 3- > 1- > 2

Advantages:

1. Increase search efficiency
2. Promotes Reusability to some extent.

Disadvantage:

1. Difficult to maintain so many links and connection.
2. Extra space requirement is there.

HASH TABLES:

1. Hashing is important technique used to search the records of symbol table.
2. In hashing two tables are maintained

(*a*) Hash table

(*b*) Symbol table

3. It consist of K entries from 0 to K-1.Thse entries are basically pointer to symbol table pointing to names of symbol table.

4. To determine whether "name" is in symbol table we, use hash function 'h'such that (Name) will result integer between 0 to K-1 we can search any name by position = h(Name).

5. Using this position we can obtain the exact location of name in symbol table.

6. Hash function should result in distribution of name in symbol table.

7. There should be minimum no. of collisions..Collisionis a situation where hash function result in same location for storing names.

8. Various collision resolution techniques are Open Addressing, Chaining, Rehashing

Advantages:

1. Quick Search

Disadvantage:

1. It is complicated to implement.
2. Extra space is required
3. Obtaining Scope of variables is difficult.

Types of Hash Function:

1. **Mid Square Method:**

(*a*) Consider a key K

Let K = 3230

(*b*) Find the square of K

$K = (3230)^2 = 10432900$

(*c*) Apply hash function

(*i*) Hash function will eliminate few digits from both sides of k2.

(*ii*) Hence H(K) = 32

Similarly

K	7236	2435
K2	52359696	5929225
H(K)	59	29

(*d*) Deleting the number of digits from both sides depend on maximum no. of keys required, deletion may or may not be same from both ends.

(2) Division Method

Let Key = K

No. of elements = *n*

Hash function = H(K)

H(K) = K mod m or **H(K) = K mod m + 1**

Where m = Prime no

Ex: We have to index 100 records and hence largest prime number close to 100 is 97 hence we will choose m = 97

Let K = 3230

H(3230) = 3230 mod 97 = **29**

3. Folding Method

(*i*) Use this method for **large key values.**

(*ii*) We can use two methods **to get hash value.**

- Shift Folding

  $K = 324\ 987\ 626\ 191\ 532$

  Divide this key **value in to parts K1, K2, K3, K4, K5**

  K1 = 327

  K2 = 987

  K3 = 626

  K4 = 191

  K5 = 532

  2663

- Boundary Folding

  In this some of **sub keys are reversed to generate new simple hash values**

K = 324 987 626 191 532

Divide this key value in to parts K1, K2, K3, K4, K5

K1 = 327

K2 = 789

K3 = 626

K4 = 191

K5 = 253

2168

4. Trees:

1. Symbol table can also be stored in form of binary search tree.
2. Each node can have atmost two children-left and right child
3. Key value stored in parent node is more than key value stored in left of children and less than right child.
4. Height of binary search tree is of the order of log n
5. Maximum time required for searching and inserting is proportional to O(log n).

Algorithm:

1. Initially, ptr will point to root of tree.
2. While ptr = Null do
3. If Name = Name(ptr)
   then return true
4. else if Name < Name (ptr) then
   Ptr: = Left(ptr)
5. elseptr : = Right(ptr)
6. end of loop.

END TERM EXAMINATION [MAY 2016]
SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS-302]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any three questions including Q. no. 1 which is compulsory.*

Q.1. Attempt all 10 questions.

Q.1. (a) Distinguish between compiler and interpreter. Define Cross Compiler and identify the few cases where such cross compiler will be useful.

Ans:

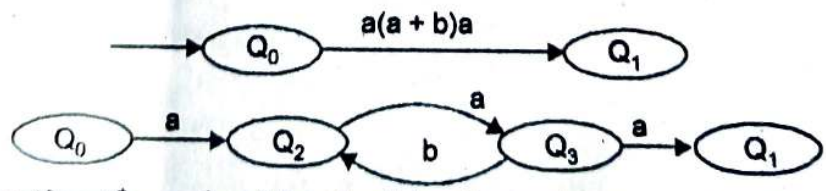
S. No.	Compiler	Interpreter
1.	Compiler takes Entire program as input	Interpreter takes Single instruction as input.
2.	Intermediate Object Code is Generated	No Intermediate Object Code is Generated
3.	Conditional Control Statements are Executes faster	Conditional Control Statements are Executes slower
4.	Memory Requirement : More (Since Object Code is Generated)	Memory Requirement is Less
5.	Program need not be compiled every time	Every time higher level program is converted into lower level program
6.	Errors are displayed after entire program is checked	Errors are displayed for every instruction interpreted (if any)
7.	Example : C Compiler	Example : BASIC

A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler. Cross compiler tools are generally found in use to generate compiles for embedded system or multiple platforms. It is a tool that one must use for a platform where it is inconvenient or impossible to compile on that platform, like microcontrollers that run with a minimal amount of memory for their own purpose. It has become more common to use this tool for para virtualization where a system may have one or more platforms in use.

The fundamental use of a cross compiler is to separate the build environment from the target environment.

Q.1. (b) Design the Finite Automata for the regular expression a(a+b)a.

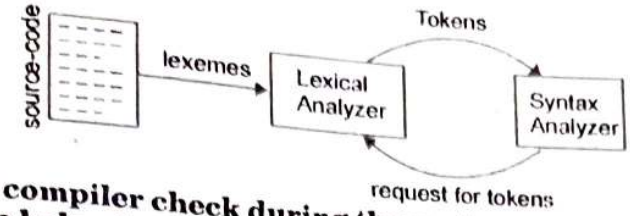
Ans:



Q.1. (c) Mention the role of Lexical Analyser in compiler design.

Ans: Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

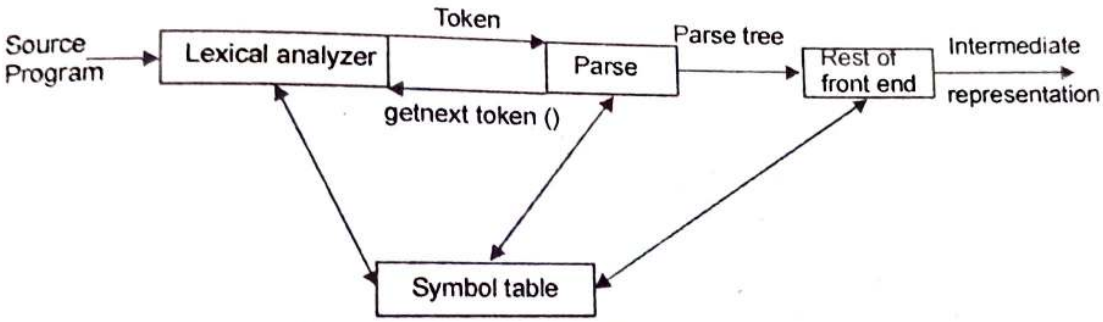


Q.1. (*d*) What compiler check during the syntax analysis? Mention the role of parser with the help of suitable figure during the syntax analysis.

Ans: It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Role of Parser

1. Performs context free syntax analysis.
2. Guides context –sensitive analysis.
3. Constructs an intermediate representation.
4. Produces meaningfull analysis.
5. Attempts Error Correction.



Q.1. (*e*) Define ambiguity in a grammar.

Ans: : If a context free grammar G has more than one derivation tree for stringw *wt*L(G), it is called an **ambiguous grammar**. There exist multiple right-most or left-most derivations for some string generated from that grammar.

Problem:

To check whether the grammar G with production rules –

$X \rightarrow X + X \mid X*X \mid X \mid a$ is ambiguous or not.

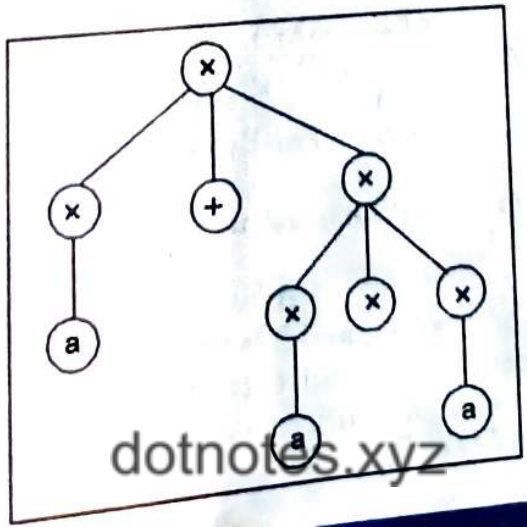
Solution:

Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

Derivation 1

$X \rightarrow X + X \rightarrow a + X \rightarrow a + X*X \rightarrow a + a*X \rightarrow a + a*a$

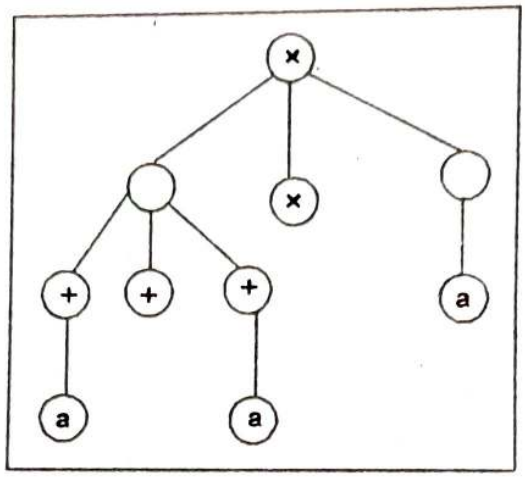
Parse tree 1 –



Derivation 2 –

$X \rightarrow X^*X \rightarrow X + X^*X \rightarrow a + X^*X \rightarrow a + a^*X \rightarrow a + a^*a$

Parse tree 2 –



As there are two parse trees for a single string "a + a*a", the grammar G is ambiguous.

Q.1. (f) What are the advantages of performing the LR parsing?

Ans: LR Parsing:

L-Scans the input from left to right.

R-Construct the right most derivation.

K-No. of Look-Ahead Symbols.

Advantages:

1. Can be constructed to recognize virtually all programming language constructs for which context free grammers can be written.
2. It is more general than other types of parsers such as LL(k),operator and simple precedence parsers and is as efficient.
3. Can detect syntactic errors as soon as it is possible to do so on a left to right scan of the input.

Q.1. (g) Define FIRST and FOLLOW sets for a grammar.

Ans: FIRST: Collection of terminals which are the start symbol of all the strings generated by a variable.

1. If X is terminal, FIRST(X) = {X}.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
3. If X is a non-terminal, and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, and ε is in all of $FIRST(Y_1), .., FIRST(Y_k)$, then add ε to FIRST(X).
4. If X is a non-terminal, and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then add a to FIRST(X) if for some i, a is in $FIRST(Y_i)$, and ε is in all of $FIRST(Y_1), .., FIRST(Y_{i-1})$.

Applying rules 3 and 4 for $FIRST(Y_1 Y_2 \ldots Y_k)$ can be done as follows:

Add all the non-ε symbols of $FIRST(Y_1)$ to $FIRST(Y_1 Y_2 \ldots Y_k)$. If $\varepsilon \in \varepsilon$ $FIRST(Y_1)$, add all the non-ε symbols of $FIRST(Y_2)$. If $\varepsilon \in FIRST(Y_1)$ and $\varepsilon \in FIRST(Y_2)$, add all the non-$\varepsilon$ symbols of $FIRST(Y_3)$, and so on. Finally, add ε to $FIRST(Y_1 Y_2 .. Y_k)$ if $\varepsilon \in FIRST(Y_i)$, for all $1 \leq i \leq k$.

FOLLOW: Collection of all the terminal symbols which immediately comes after the strings generated by B

$$A \rightarrow \alpha B\beta$$

1. If $ is the input end-marker, and S is the start symbol, $ ∈ FOLLOW(S).
2. If there is a production, $A \rightarrow \alpha B\beta$, then $(FIRST(\beta) - \varepsilon) \subseteq FOLLOW(B)$.
3. If there is a production, $A \rightarrow \alpha\beta$, or a production $A \rightarrow \acute{a}B\varepsilon$, where $\varepsilon \in FIRST(\beta)$, then $FOLLOW(A) \subseteq FOLLOW(B)$.

Q.1. (*h*) Perform recursive descent parsing over the input mpp for the grammar.

$$S \rightarrow XY$$
$$X \rightarrow m \mid mY$$
$$Y \rightarrow p$$

Ans: boolean match (token tok){return * next++ = tok;}
boolean S ( ) { return X( ) & return Y ( );}
boolean X( ) { return match(b) & return match (b) & return Y( ) ;}
boolean Y ( ) { return match( );}

```
boolean S ( ) { token * save = next; return ( next = save, S1 ( ) )|
boolean X ( ) { token * save = next; return ( next = save, X1 ( ) )|
| | ( next = save X2( ) ) }
boolean Y ( ) { token * save = next; return ( next = save,Y1 ( ) )|
```

Q.1. (*i*) What is Peephole optimization? How is it performed? Give example of Peephole optimization.

Ans: Peephole optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window". It works by recognising sets of instructions that can be replaced by shorter or faster sets of instructions. Common techniques applied in peephole optimization:

- Constant folding – Evaluate constant subexpressions in advance.
- Strength reduction – Replace slow operations with faster equivalents.
- Null sequences – Delete useless operations.
- Combine operations – Replace several operations with one equivalent.
- Algebraic laws – Use algebraic laws to simplify or reorder instructions.
- Special case instructions – Use instructions designed for special operand cases.
- Address mode operations – Use address modes to simplify code.

Redundant instruction elimination

At source code level, the following can be done by the user:

```
int add_ten(int x)
{
int y, z;
 y =10;
 z = x + y;
return z;
}
```

```
int add_ten (int x)
{
int y;
 y =10;
 y = x + y;
return y;
}
```

```
int add_ten(int x)
{
int y = 10;
return x + y;
}
```

```
int add_ten(int x)
{
return x +10;
}
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

- MOV x, R0
- MOV R0, R1

We can delete the first instruction and re-write the sentence as:

MOV x, R1

Unreachable code

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

Example:

```
void add_ten(int x)
{
return x +10;
```

```
    printf("value of x is%d", x);
}
```

In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

Flow of control optimization

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
...
MOV R1, R2
GOTO L1
...
L1 : GOTO L2
L2 : INC R1
```

In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...
MOV R1, R2
GOTO L2
...
L2 : INC R1
```

Algebraic expression simplification

There are occasions where algebraic expressions can be made simple. For example, the expression a = a + 0 can be replaced by a itself and the expression a = a + 1 can simply be replaced by INC a.

Strength reduction

There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result.

For example, **x * 2** can be replaced by **x << 1**, which involves only one left shift. Though the output of a * a and a^2 is same, a^2 is much more efficient to implement.

Accessing machine instructions

The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently. If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient result.

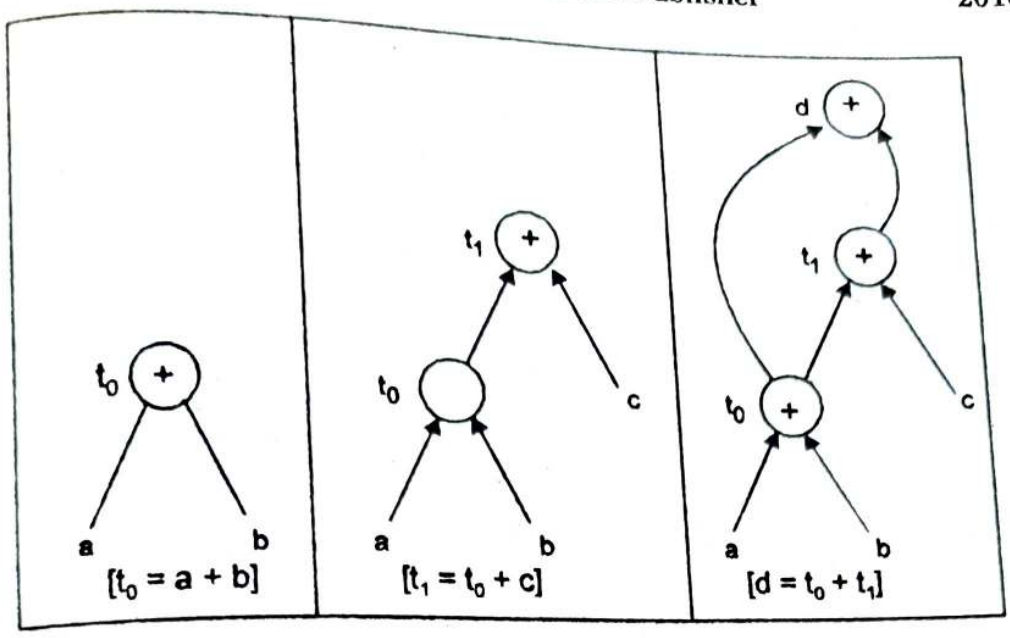
Q.1. (j) What is DAG? What are the advantages of using DAG?

Ans: Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

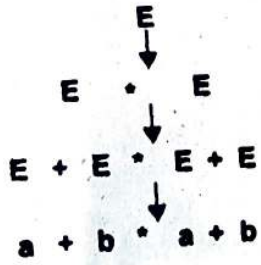
- $t_0 = a + b$
- $t_1 = t_0 + c$
- $d = t_0 + t_1$



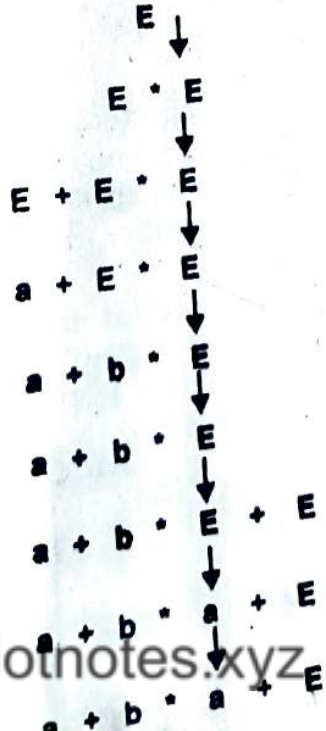
UNIT-I

Q.2 (a) Design a derivation tree for the following grammar: E → E+E | E-E | E/E | E*E | a | b. Also obtain the left most and right most derivation tree for the string "a + b * a + b".

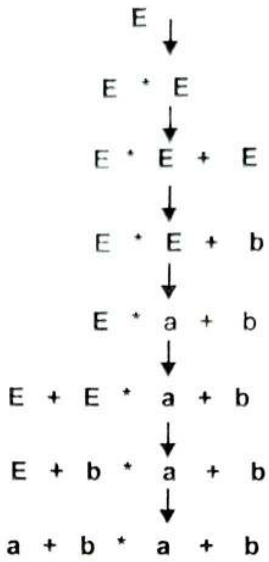
Ans: Derivation Tree



Left Most Derivation Tree



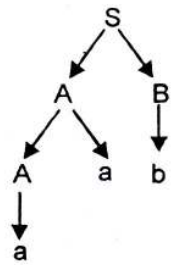
Right Most Derivation Tree



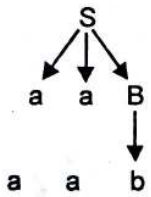
***b*) Find whether the grammar is ambiguous or not:**

S AB/aaB, A → a/Aa,B → b How to remove ambiguity from this grammar

Ans: (1)



(2)



Removal of Ambiguity:

1. Precedence of **Operators is not respected.**

2. Sequence of **identical operators can group either from left or from right.** We would see two different parse **tree for the above expression. Since addition is associative, it** doesn't matter whether **the group is from left or right, but to eliminate the ambiguity we** must take one.

***Q.3 (a)* Consider the following grammar and Parse the input string int id, id; using shift reduce parser.**

S → TL; T int | float L L, id | id →

Ans:

STACK	INPUT STRING	ACTION
$	int id, id;	Shift
$int	id, id;	Reduce by T -> int
$ T id	,id;	Shift
$ T L	,id;	Reduce by L - > L

$ T L,	id;	Shift
$ T L, id	;	Shift
$ T L	;	Replace L ->L,id
$ T L;	$	Replace S -> TL;
$ S	$	Accept

3. (*b*) Discuss all three methods which are used to perform the LR parsing. Which technique is most powerful among these three? Give justification also.

Ans: LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of look ahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

- SLR(1) – Simple LR Parser:
  - Works on smallest class of grammar
  - Few number of states, hence very small table
  - Simple and fast construction
- CLR(1) – Canonical LR Parser:
  - Works on complete set of LR(1) Grammar
  - Generates large table and large number of states
  - Slow construction
- LALR(1) – Look-Ahead LR Parser:
  - Works on intermediate size of grammar
  - Number of states are same as in SLR(1)

Difference between SLR(1), LR(1) and LALR(1):

S.NO.	SLR PARSER	LALR PARSER	CLR PARSER
1.	Easy and cheap to implement	Easy and cheap to implement	Expensive and difficult to implement
2.	Smallest in size	LALR and SLR have same size, as they have less no of states	CLR parser is largest in size, as no. of states are very large.
3.	Error detection is not immediate	Error detection is not immediate	Error detection can be done immediately
4.	SLR fails to produce parsing table for certain class of grammars	Intermediate in power between SLR nad CLR	Very powerful and works on large class of grammers
5.	Requires less time and space complexity	Requires more time and space complexity	Requires more time and space complexity

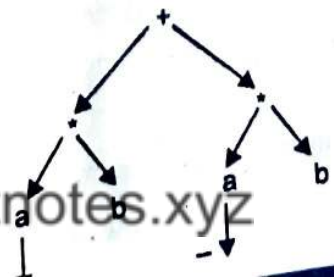
UNIT–II

Q.4. What are the benefits for the three address code generation? Consider the input string.

x: = – a * b + – a * b and generate the following:

(*a*) Syntax Tree (*b*) Postfix (*c*) Three address Code

Ans: (*a*) Syntax Tree:



(b) Postfix: a – b*a – b*+

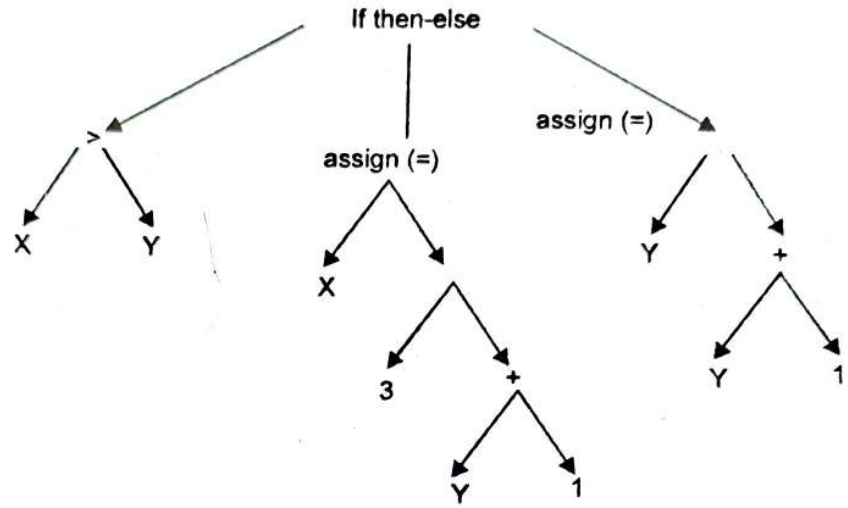
(c) Three Address Code: x: = – a * b + – a * b

t1 = – a
t2 = t1*b
t3 = t1 * b
t4 = t1 + t3

Q5. (a) Draw the Syntax tree for the following piece of code in the source language

If X > Y then X = 3 * (Y+ 1) else Y = Y + 1

Ans:



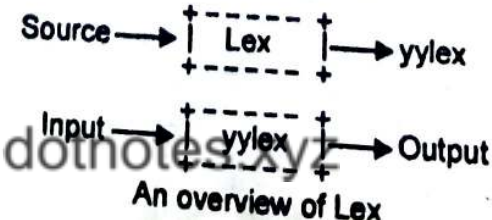
Q.5. (b) Write short comments on:

(i) LEX **(ii) YACC**

Ans: (i) LEX: Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected.



An overview of Lex

(ii) **YACC:**

YACC Specifications

- **Similar structure to LEX**
  - Declaration Section
  - Translation Rules
  - Supporting C/C ++ code

Declaration Section: – C/C ++ Code –

YACC definition

- %token
- %start
- Others

The rules section represents a grammar. The left-hand side of a production is followed by a colon. Actions associated with a rule are entered in braces.

Structure of a yacc: A yacc file looks much like a lex

file: ...definitions... %% ...rules... %% ...code...

Definitions As with lex, all code between %{ and %} is copied to the beginning of the resulting C file.

As with lex, a number of combinations of pattern and action. The patterns are now those of a context-free grammar, rather than of a regular grammar as was the case with lex. code. This can be very elaborate, but the main ingredient is the call to yyparse, the grammatical parse.

Q.6. What is the significance of the symbol table in compiler? What information is represented by symbol tables? Explain the data structure used for symbol table.

Ans: Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

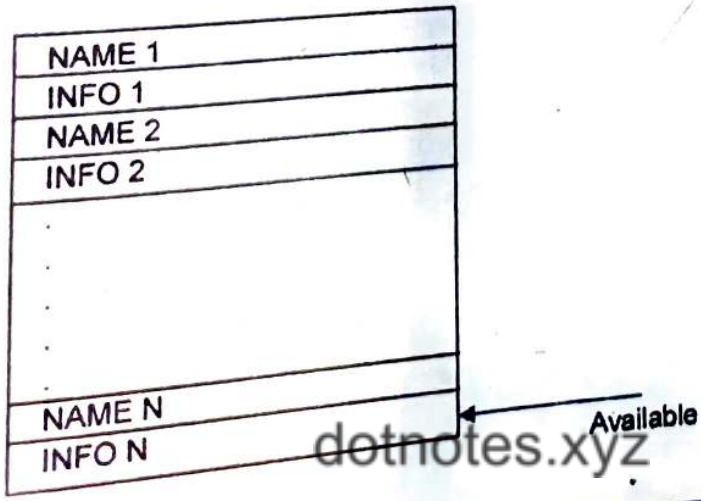
- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

Different data structures used for symbol table are:

1. Lists
2. Self Organising Lists
3. Trees
4. Hash Tables

LISTS:

1. It is simple and easy to implement data structure for symbol table in linear list records:



2. One can use single array to store the name and its associated information.
3. New names are added to end of list on FCFS basis.
4. Method of access is sequential

Advantages:

1. Minimum soace requirement.
2. Easy to understand and implement.

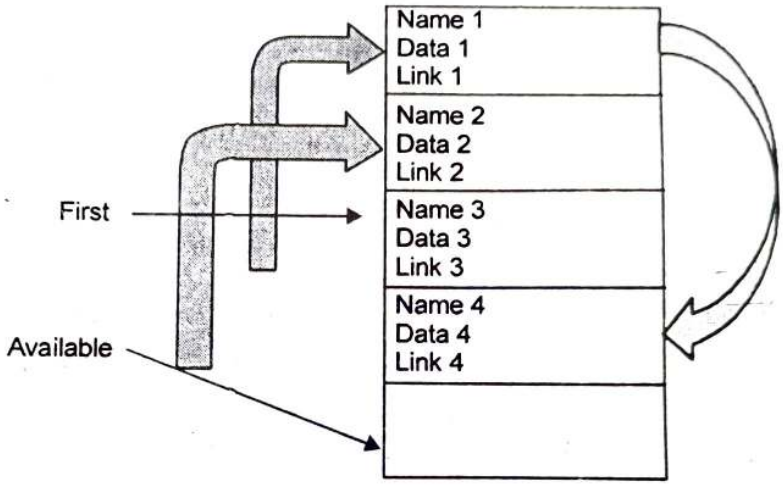
Disadvantages:

1. Sequential Access.
2. Amount of work done required to be done is high.

SELF ORGANISING LISTS:

1. It is used to decrease searching time in list organisation.

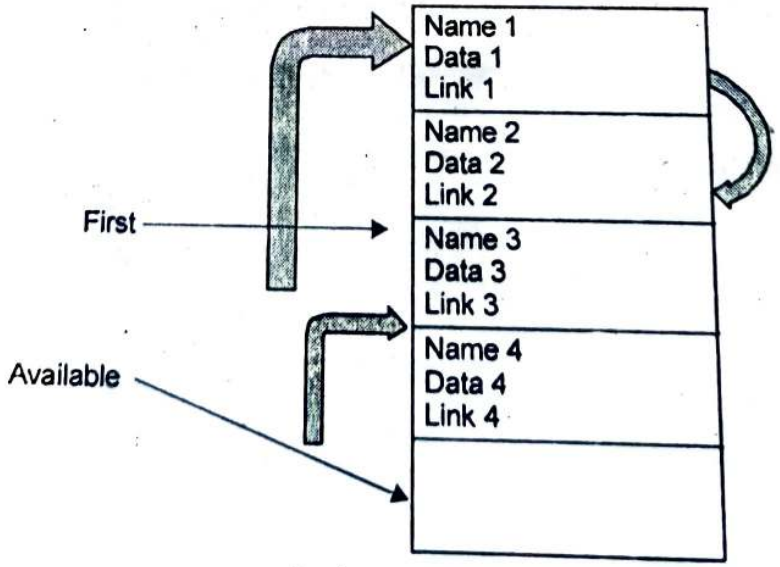
2. Special attribute link is added to information of name that allows dynami features in list, this feature help in minimizing wastage of space and also promote reusability to some extent.



– First pointer points to the beginning of symbol table
– If we have names in the following order

3 -> 1 -> 4 -> 2

– If we need a particular record we need to move record to the front of the list. Suppose we need 4, so new connection will be



Mechanism

(*a*) Initial stage 3 -> 1 -> 4 -> 2

(*b*) 4^{th} name is required

(*c*) move 4^{th} to front

3 -> 1 -> **4** -> 2

(*d*) **4** -> 3 -> 1 -> 2

Advantages:

1. Increase search efficiency.
2. Promotes Reusability to some extent.

Disadvantage:

1. Difficult to maintain so many links and connection.
2. Extra space requirement is there.

HASH TABLES:

1. Hashing is important technique used to search the records of symbol table.
2. In hashing two tables are maintained.

(*a*) Hash table

(*b*) Symbol table

3. It consist of K entries from 0 to K-1. These entries are basically pointer to symbol table pointing to names of symbol table.

4. To determine whether "name" is in symbol table we, use hash function 'h' such that (Name) will result integer between 0 to K-1 we can search any name by position = h(Name).

5. Using this position we can obtain the exact location of name in symbol table.

6. Hash function should result in distribution of name in symbol table.

7. There should be minimum no. of collisions.Collision is a situation where hash function result in same location for storing names.

8.Various collision resolution techniques are Open Addressing, Chaining, Rehashing.

Advantages:

1. Quick Search

Disadvantage:

1. It is complicated to implement.
2. Extra space is required
3. Obtaining Scope of variables is difficult.

Types of Hash Function:

(1) **Mid Square Method:**

(*a*) Consider a key K

Let K = 3230

(*b*) Find the square of K

$K^2 = (3230)^2 = 10432900$

(*c*) Apply hash function

(*i*) Hash function will eliminate few digits from both sides of k2.

(*ii*) Hence H(K) = 32

Similarly

K	7236	**2435**
K2	52359696	**5929225**
H(K)	59	**29**

(*d*) Deleting the number of digits from **both sides depend on maximum no. of keys** required, deletion may or may not be same **from both ends.**

(2) Division Method

Let Key = K

No. of elements = n

Hash function = H(K)

H(K) = K mod m or H(K) = K mod m + 1

Where m = Prime no.

Ex: we have to index 100 records and hence largest prime number close to 100 is 97 hence we will choose m = 97

Let K = 3230

H(3230) = 3230 mod 97 = 29

(3) Folding Method

(*i*) Use this method for large key values.

(*ii*) We can use two methods to get hash value.

- Shift Folding

  K = 324 987 626 191 532

  Divide this key value in to parts K1, K2, K3, K4, K5

  K1 = 327

  K2 = 987

  K3 = 626

  K4 = 191

  K5 = 532

  2663

- Boundary Folding

  In this some of sub keys are reversed to generate new simple hash values

  K = 324 987 626 191 532

  Divide this key value in to parts K1, K2, K3, K4, K5

  K1 = 327

  K2 = 789

  K3 = 626

  K4 = 191

  K5 = 253

  2168

4. Trees:

1. Symbol table can also be stored in form of binary search tree.
2. Each node can have atmost two children-left and right child.
3. Key value stored in parent node is more than key value stored in left of children and less than right child.
4. Height of binary search tree is of the order of log n.
5. Maximum time required for searching and inserting is proportional to O(log n).

Algorithm:

(1) Initially, ptr will point to root of tree.

(2) While ptr = Null do

(3) If Name = Name(ptr)

then return true

(4) else if Name < Name(ptr) then

Ptr:= Left(ptr)

(5) else ptr : = Right(ptr)

(6) end of loop.

Q7. What is activation Record? Briefly outline Run time storage administration. Discuss different error handling techniques.

Ans: A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.

The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e., parameters which are used to send input to the called procedure.
Return Value	Stores return values.

Storage Allocation Runtime environment manages runtime memory requirements for the following entities:

Code : It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.

Procedures : Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.

Variables : Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Static Allocation in this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required. Stack Allocation Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out LIFO method and this allocation strategy is very useful for recursive procedure calls.

Heap Allocation Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required. Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.

The text part of the code is allocated a fixed amount of memory. Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both shrink and grow against each other.

Error Handling Techniques: A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- **Lexical** : Name of some identifier typed incorrectly.
- **Syntactical** : Missing semicolon or unbalanced parenthesis.
- **Semantical** : Incompatible value assignment.
- **Logical** : Code not reachable, infinite loop.

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-

colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

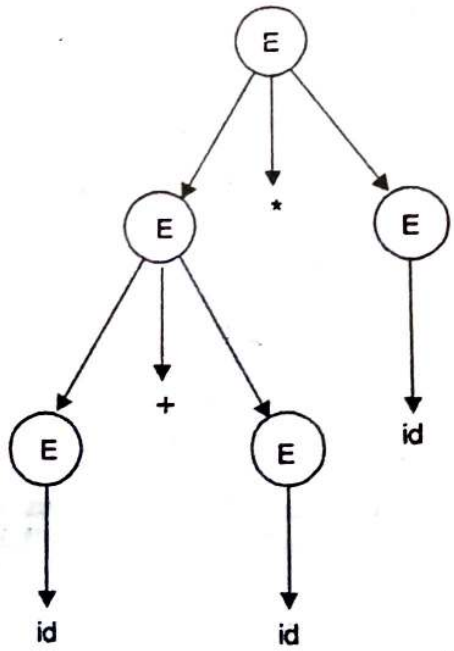
Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

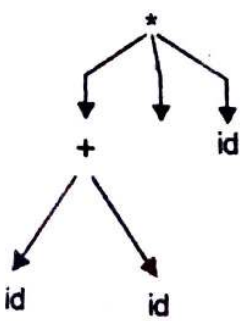
The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Abstract Syntax Trees

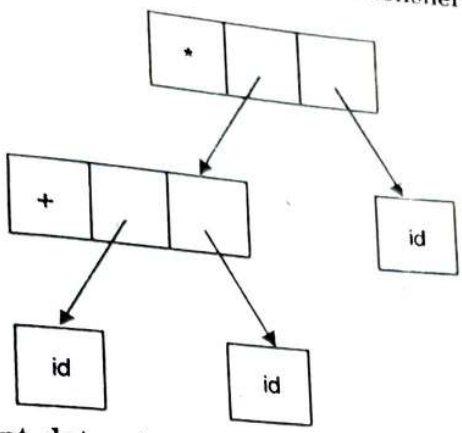
Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed. Take the following parse tree as an example:



If watched closely, we find most of the leaf nodes are single child to their parent nodes. This information can be eliminated before feeding it to the next phase. By hiding extra information, we can obtain a tree as shown below:



Abstract tree can be represented as:



ASTs are important data structures in a compiler with least unnecessary information. ASTs are more compact than a parse tree and can be easily used by a compiler.

UNIT-IV

Q.8. What are the criteria that need to be considered while applying the code optimization technique? Mention the issues involved in designing of code generation.

Ans. Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types : machine independent and machine dependent.

Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
  item =10;
  value = value + item;
} while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item=10;
do
{
value = value + item;
}while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

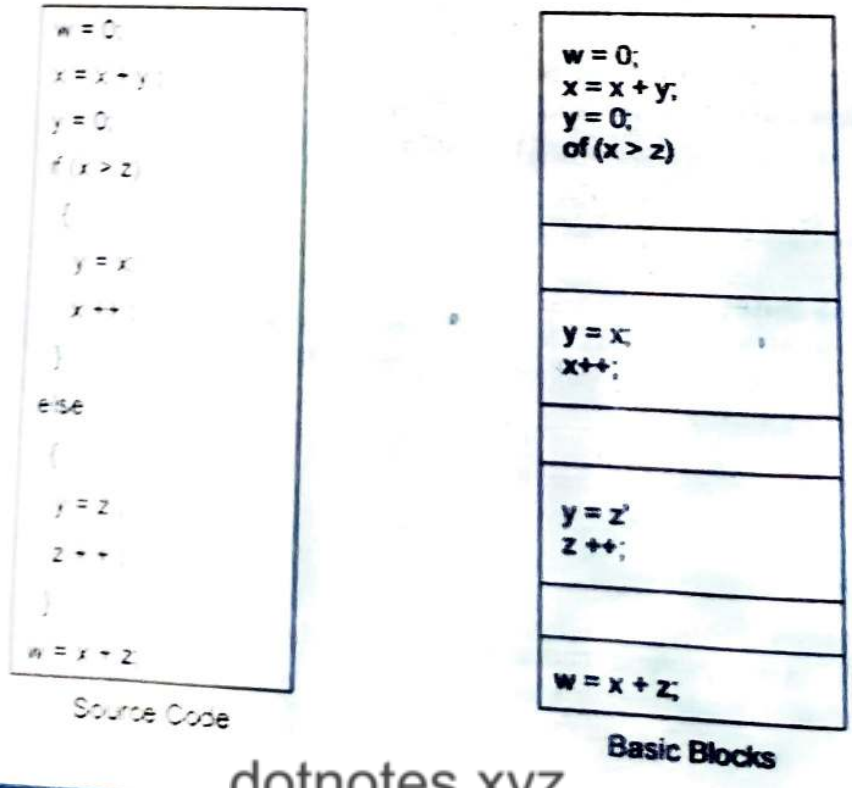
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic block identification

We may use the following algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:
  - First statement of a program.
  - Statements that are target of any branch (conditional/unconditional).
  - Statements that follow any branch statement.
- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

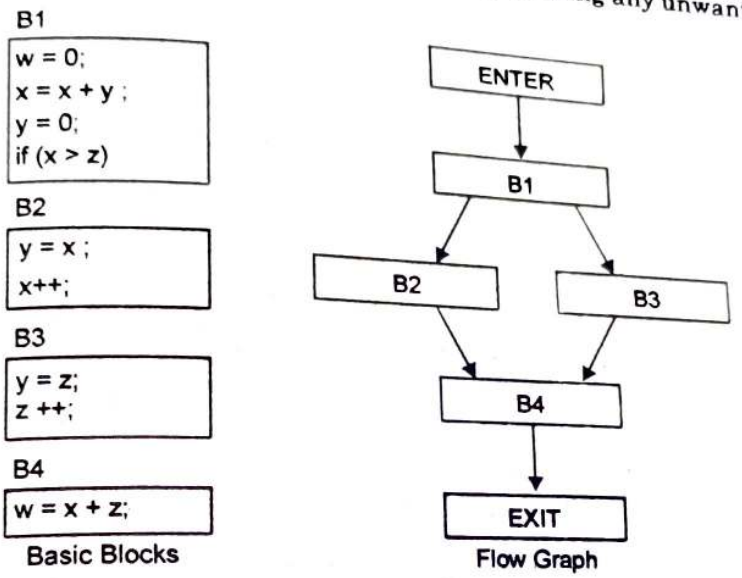
Basic blocks are important concepts from both code generation and optimization point of view.



Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.



Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques.

- **Invariant code** : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.
- **Induction analysis** : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.
- **Strength reduction** : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

Dead-code Elimination

Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

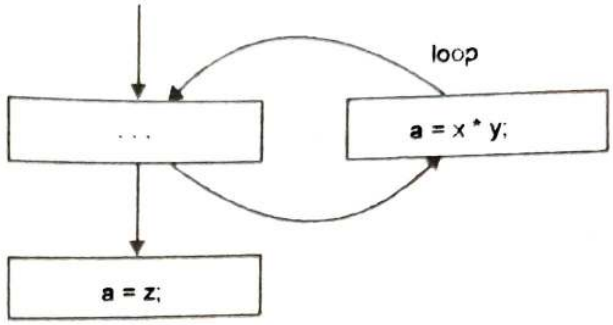
Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Partially dead code

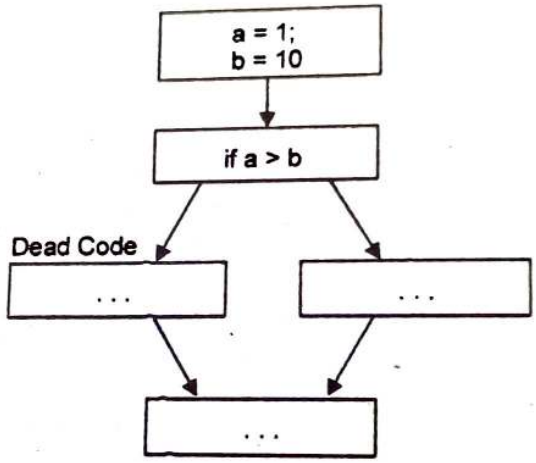
There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.

The above control flow graph depicts a chunk of program where variable 'a' is used to assign the output of expression x * y. Let us assume that the value assigned to 'a' is

never used inside the loop.Immediately after the control leaves the loop, 'a' is assigned the value of variable 'z', which would be used later in the program. We conclude here that the assignment code of 'a' is never used anywhere, therefore it is eligible to be eliminated.

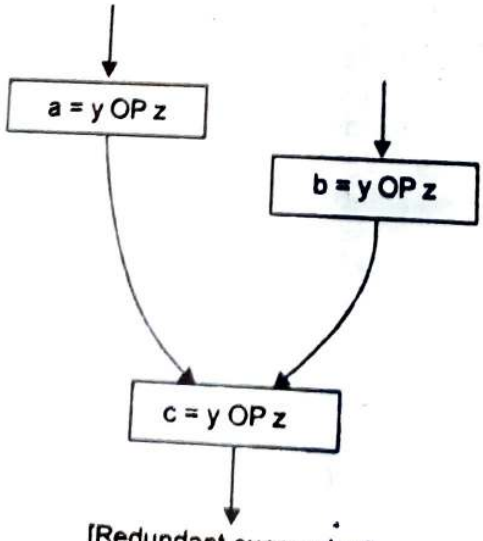


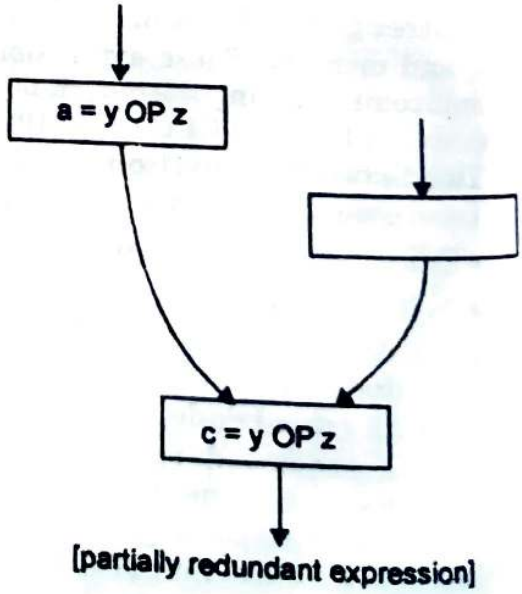
Likewise, the picture above depicts that the conditional statement is always false implying that the code, written in true case, will never be executed, hence it can be removed.



Partial Redundancy:

Redundant expressions are computed more than once in parallel path, without any change in operands.whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,





Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique.

Another example of a partially redundant code can be:

```
If(condition)
{
a = y OP z;
}
else
{
...
}
c = y OP z;
```

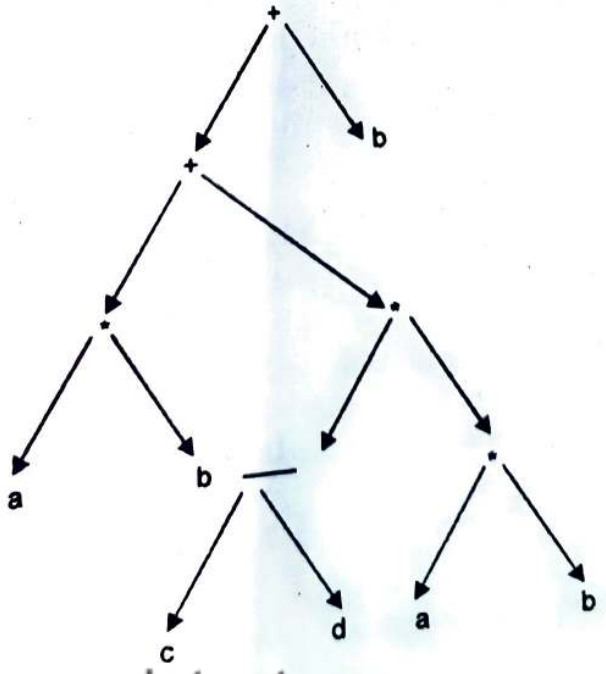
We assume that the values of operands (y and z) are not changed from assignment of variable a to variable c. Here, if the condition statement is true, then y OP z is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

```
If(condition)
{
...
    tmp = y OP z;
    a = tmp;
...
}
else
{
...
    tmp = y OP z;
}
c = tmp;
```

Here, whether the condition is true or false; y OP z should be computed only once.

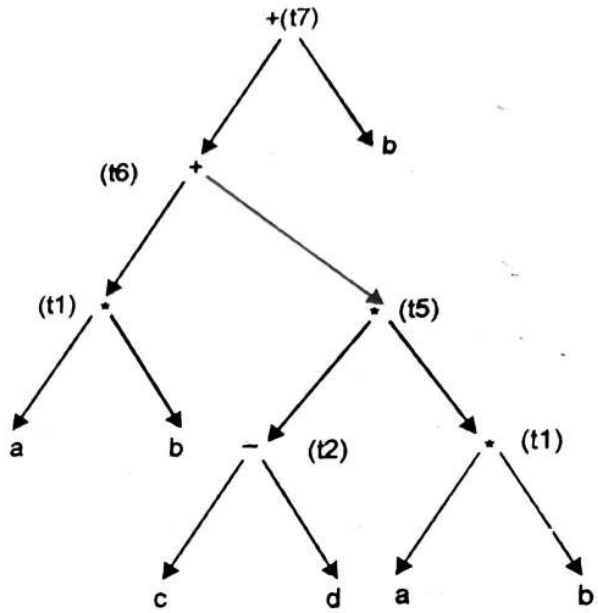
Q.9. Draw the syntax tree and DAG for the expression (a*b) +(c – d) *(a * b) + b.

Ans. Syntax tree:



DAG:

```
t1 = a * b
t2 = c * d
 t = + b
t5 = t1 * t2
t6 = t5  + t1
t7 = t6 + t3
```



FIRST TERM EXAMINATION [FEB. 2017]
SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS–302]

Time : 1.5 Hrs. M.M. : 30

Note: Attempt three questions. Question No. 1. is compulsory. Each question carries 10 marks.

Q.1. (a) Mention the role of Lexical Analyzer in compiler Design. (2)

Ans. The main task is to read the input characters and produce as output sequence of tokens that the parser uses for syntax analysis.

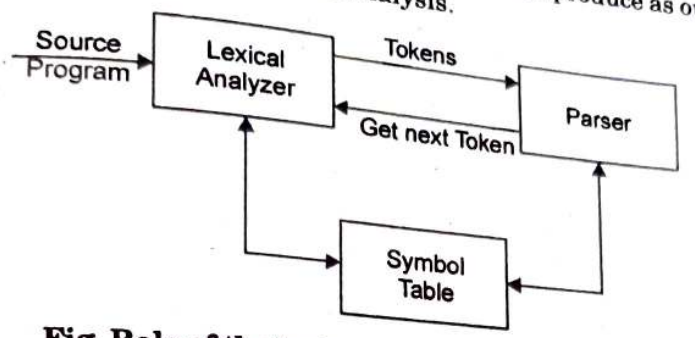


Fig. Role of the lexical analyzer diagram

Up on receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Its secondary tasks are:

- One task is stripping out from the source program comments and white space is in the form of blank, tab, new line characters.
- Another task is correlating error messages from the compiler with the source program.

Sometimes lexical analyzer is divided in to cascade of two phases.

(1) Scanning (2) Lexical analysis.

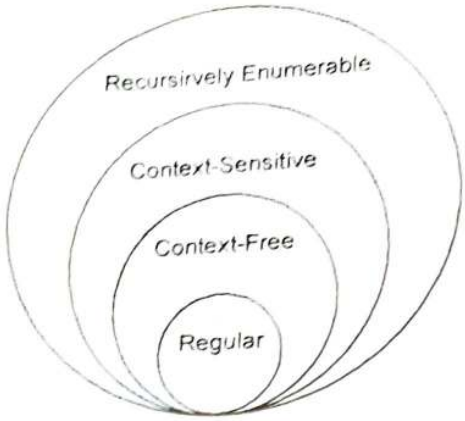
The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

Q.1. (b) Explain Chomsky classification of languages. (2)

Ans. According to Noam Chomosky, there are four types of grammars " Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other –

Grammar Type	Grammar Accepted	Language Accepted	Automaton
1. Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
2. Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
3. Type 2	Context-free grammar	Context-free language	Pushdown automaton
4. Type 3	Regular grammar	Regular language	Finite state automaton

Take a look at the following illustration. It shows the scope of each type of gramm



Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form **X →a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

X→ ε

X → a | aY

Y→ b

Type - 2 Grammar

Type-2 grammars generate context-free languages.

The productions must be in the form **A → γ**

where **A ∈ N** (Non terminal)

and **γ ∈ (T ∪ N)*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example

S → X a

X→ a

X → aX

X → abc

X→ ε

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

α A β → α γ β

where **A ∈ N** (Non-terminal)

and **α, β, γ ∈ (T ∪ N)*** (Strings of terminals and non-terminals)

The strings α and β may be empty, but γ must be non-empty.
The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

$AB \rightarrow AbBc$

$A \rightarrow bcA$

$B \rightarrow b$

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of $\alpha \rightarrow \beta$ where α is a string of terminals and nonterminals with at least one non-terminal and α cannot be null. β is a string of terminals and non-terminals.

Example

$S \rightarrow ACaB$

$Bc \rightarrow acB$

$CB \rightarrow DB$

$aD \rightarrow Db$

Q.1. (c) What are the three methods to construct LR Parsing tables.Which method is the most powerful? Justify. **(2)**

Ans. There are three widely used algorithms available for constructing an LR parser:

- SLR(1) - Simple LR
  - Works on smallest class of grammar.
  - Few number of states, hence very small table.
  - Simple and fast construction.
- LR(1) - LR parser
  - Also called as Canonical LR parser.
  - Works on complete set of LR(1) Grammar.
  - Generates large table and large number of states.
  - Slow construction.
- LALR(1) - Look ahead LR parser
  - Works on intermediate size of grammar.
  - Number of states are same as in SLR(1).

Q.1. (d) An ambiguous grammar can never be LR .Justify the statement with an example. **(2)**

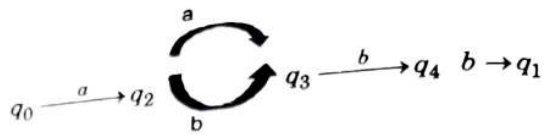
Ans: An ambiguous grammar can never be LR(k) for any k, because LR(k) algorithm aren't designed to handle ambiguous grammars. It would get stuck into undecidability problem, if employed upon an ambiguous grammar, no matter how large the constant k is.

Q.1. (e) Design the NFA for the regular Expression a (a+b)* bb **(2)**

Ans.

$$q_0 \underline{a(a+b)^* bb}\ q_1$$

$$q_0 \underline{a}\ q_{2(a+b)}{}^* q_3 \underline{b}\ q_4 \underline{b}\ q_1$$



Q.2. (a) Let G be CFG:

$$S \rightarrow bB \mid aA$$
$$A \rightarrow b \mid bS \mid aAA$$
$$B \rightarrow a \mid aS \mid bBB$$

For the string bbaababa find

(i) Left Most Derivative

(ii) RightMost Derivative

(iii) Parse Tree

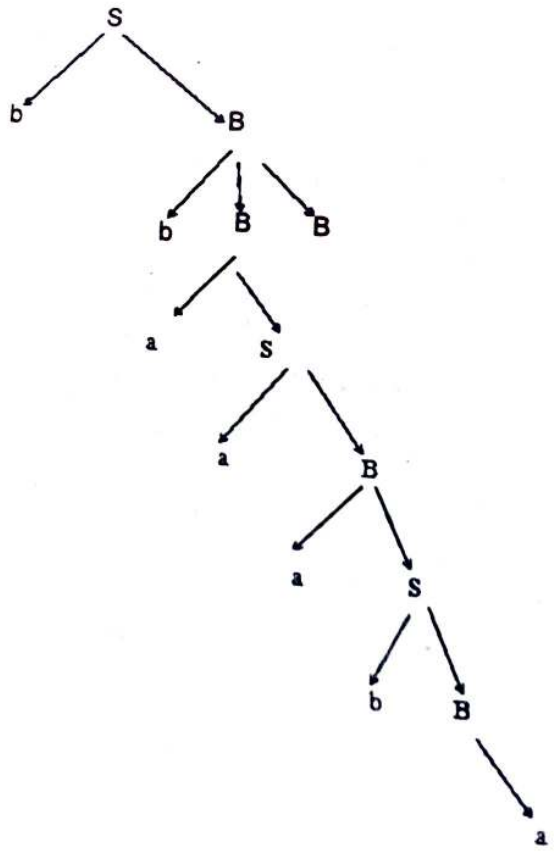
Ans. (i) Left Most Derivative:

$S \rightarrow bB$
bbBB
bbaSB
bbaaB
bbaabBB
bbaabaSB
bbaababa

(ii) RightMost Derivative: $S \rightarrow bB$

bbBB
bbBa
bbaSa
bbaaAa
bbaabSa
bbaabaAa
bbaababa

(iii) Parse tree



Q.2. (b) Construct predictive parsing table for the following grammar(S is Start Symbol) (5)

$$S \rightarrow i\,C\,t\,SS' \mid a$$
$$S' \rightarrow eS \mid \varepsilon$$
$$C \rightarrow b$$

Ans. Step 1. Eliminate left-Recursion and Left Factor the grammer if required
As, there is no left. Recursion and grammar is already left Factored.

(5)

Step 2. Computation of FIRST

(i) $S \rightarrow i\,C\,t\,SS' \mid a$

$\therefore$ FIRST (S) = {i, a}

(ii) $S' \rightarrow e\,S \mid \varepsilon$

$\therefore$ FIRST (S') = {e, ε)

(iii) $C \rightarrow b$

$\therefore$ FIRST (C) = {b}

FIRST (S) = {i, a}
FIRST (S')= {e, ε}
FIRST (C) = {b}

Step 3. Computation of FOLLOW:
Applying Rule (1) of FOLLOW

FOLLOW (S) = {$}

(1) $S \rightarrow i\,C\,t\,S\,S'$

Applying Rule (2) of FOLLOW.

$S \rightarrow$	i	C	tSS'
$A \rightarrow$	α	B	β

$\therefore$ FIRST (β) = FIRST (t SS') = {t}.
$\therefore$ FIRST (β) does not contain ε.
$\therefore$ By Rule (2a) FOLLOW (C) = {FIRST (t SS')}

$\therefore$ FOLLOW (C) = [t]

Again, Apply Rule (2) on different combination

$S \rightarrow$	iCt	S	S'
$A \rightarrow$	α	B	β

FIRST (β) = FIRST (S') = {e, ε} contain ε.
By Rule (2b) FOLLOWS (S) = FIRST (S') – {ε} ∪ FOLLOW (S)
FOLLOW (S) = {e, ε} – {ε} ∪ FOLLOWS (S)

FOLLOW (S) = {e} ∪ FOLLOW (S)

Applying Rule (3) of Follow

$S \rightarrow$	$iCtS$	S'
$A \rightarrow$	α	B

A = S, α = i C t S, B = S'

$\therefore$ FOLLOW (S') = {FOLLOW (S)}

(2) $S' \rightarrow eS$

Rule (2) can not be applied on this Production.

Applying Rule (3) of FOLLOW

$S' \rightarrow$	e	S
$A \rightarrow$	α	B

FOLLOW (S) = {FOLLOW (S')}

The Renaming Production $S \rightarrow a$, $S' \rightarrow \varepsilon$ and $C \rightarrow b$ do not match with an Rule

Combining Statements (1) to (5)

FOLLOWS (S) = {\$}
FOLLOWS (C) = {t}
FOLLOW (S) = {e} $\cup$ FOLLOW (S)
FOLLOW (S') = {FOLLOW (S)}
FOLLOW (S) = {FOLLOW (S')}

$\therefore$ FOLLOW (S) = FOLLOW (S)={\$; e}
FOLLOW (C) = {t}

Step 4: **Construction of Predictive Parsing Table**. Put all Non- terminals S'C'S Row wise and All terminals. i, a, b, e, \$, t Column wise

(1) $S \rightarrow i\ C\ t\ SS'$

Comparing $S \rightarrow i\ C\ t\ S\ S'$ with $A \rightarrow \alpha$

$S \rightarrow$	$iCtSS'$
$A \rightarrow$	α

FIRST (α) = FIRST (i C t S S') = {i}

Applying Rule (1) of predictive Parsing Table:-

Add $S \rightarrow iCtSS'$ to M[S, i]

Write $S \rightarrow$ i C t S S' in front of Row (S) and Column (i) ...(1)

(2) $S \rightarrow a$

Comparing it with $A \rightarrow \alpha$

$S \rightarrow$	a
$A \rightarrow$	α

$\therefore$ A = S, α = a

FIRST (α) = FIRST (a) = {a}

$\therefore$ Put $S \rightarrow a$ onto M[S, a]

Write $S \rightarrow$ a in front of Row (s), column (a)

(3) $S' \rightarrow eS$

Compairing it with $A \rightarrow \alpha$

$S' \rightarrow$	eS
$A \rightarrow$	α

∴ A = S', α = eS
∴ FIRST (α) = FIRST (eS) = {e}
∴ Add S' → eS to M [S', e]

Write S' → eS in front of Row (S'), column (e)

(4) S' → ε
Comparing it with

A → α

S' →	ε
A →	α

∴ A = S', α = ε
∴ FIRST (α) = FIRST (ε) = {ε}
∴ Applying Rule (2) of Predictive Parsing Table. i.e. If ε is in FIRST (α)
Then FOLLOW (S') = {e, $}
∴ Add S' → ε to M [S', e] M [S,' $]

write S' →ε in front of Row (S') Column {(e, $)}

(5)
Comparing it with

C → b
A → α

C →	b
A →	α

∴ First (α) = FIRST (b) = {b}
∴ Add C → b to M [C, b]

write C → b in front of Row (C), Column (b)

∴ Combining statements (1) to (5) we get following Predictive Parsing table.

	a	b	e	i	t	$
S	S → a			S → i C t S S'		
S'			S' → eS S' → ε			S' → ε
C		C → b				

This Grammer is not LL (1) because In Row (S') and column (e) we have multiple Entries for productions

S' → eS
S' → ε

S' → eS appears as FIRST (S') = {e}
S' → ε appears as FOLLOW (S') = {ε}

This Grammer is not LL (1) Grammer.

Q.3. (a) Construct SLR parsing Parsing table for the following grammar: **(5)**

S → xAy | xBy | xAz
A → aS | q
B → q

Ans: Step 1: Construct augmented Grammar

(0) S' → S
(1) S → **xAy**
(2) S → **xBy**
(3) S → **xAz**
(4) A → aS
(5) B → q

Step 2. Find Closure & goto functions to construct LR (0) items. Here Boxes represent New States and Circles represent the repeating state.

Closure (S' → • S)

Io:
$S' \to \bullet S$
$S \to \bullet x\,A\,y$
$S \to \bullet x\,B\,y$
$S \to \bullet x\,A\,z$

$I_4 = \text{goto}(I_2, B)$: $S \to x\,B \bullet y$

$I_8 = \text{goto}(I_4, y)$: $S \to x\,By \bullet$

$I_1 = \text{goto}(I_0, S)$: $S' \to S \bullet$

I_5: goto (I_2, q)
$A \to q \bullet S$
$A \to q \bullet$
$B \to q \bullet$
$S \to \bullet x\,A\,y$
$S \to \bullet x\,B\,y$
$S \to \bullet x\,A\,z$

$I_9 = \text{goto}(I_5, S)$: $A \to qS \bullet$

$I_2 = \text{goto}(I_0, x)$
$S \to x \bullet Ay$
$S \to x \bullet By$
$S \to x \bullet Az$
$A \to \bullet\ qS$
$A \to \bullet\ q$
$B \to \bullet\ q$

$I_6 = \text{goto}(I_5, y)$: $S \to x\,A\,y \bullet$

$I_2 = \text{goto}(I_5, x)$
$S \to x \cdot Ay$
$S \to x \cdot By$
$S \to \cdot Az$
$A \to \cdot q$
$B \to \cdot q$

$I_3 = \text{goto}(I_2, A)$
$S \to X\ A.Y$
$S \to X\ A.Z$

$I_7 = \text{goto}(I_3, z)$: $S \to x\,A\,z \bullet$

Step 3. Computation of FOLLOW

1. $S \to x\,A\,y$

Follow (S) = {$}

Applying Rules (2a) of FOLLOW

(Comparing $S \to x\ a\ y$ with $A \to \alpha\ B\ \beta$

$\therefore$ FIRST (β) = FIRST (y) = {y}

$\therefore$ FOLLOW (A) = {y}

Rule (3) cannot be applied.

As, $S \to x\,A\,y$ cannot be compared with $A \to \alpha\ B$

2. $S \to x\ B\ Y$

Applying Rule (2a) of FOLLOW

comparing $S \to x\ By$ with $A \to \alpha\ B\ \beta$

$\therefore$ FIRST

(β) = {y}

$\therefore$ FOLLOW(B) = {y}

Rule (3) cannot be applied.

3. $S \rightarrow xAy$

Applying Rule (2a) of Follow

First (β) = {z}

$\therefore$ FOLLOW(A) = {z}

Rule (3) cannot be applied.

4. $A \rightarrow qS$

Rule (2a) cannot be applied. As $A \rightarrow qS$ cannot be compared with $A \rightarrow \alpha B \beta$

Appling Rule (3)

Comparing $A - qS$ with $A \rightarrow \alpha\beta$

$\therefore$ $A = A, \alpha = q, B = S$

$\therefore$ FOLLOW (S) = {FOLLOW (A)}

Rule (2a) and Rule (3) of FOLLOW cannot be applied on production $A \rightarrow q$ and $B \rightarrow q$. Combining statements 1 to 5

FOLLOW (A) = {y, z}
FOLLOW (s) = {$, y, z}
FOLLW (B) = {y}

Step 4: Construction of SLR (1) Parsing Table:

States	Action					goto		
	x	y	z	q	$	S	A	B
0	s2	r3/r4				1		
1					accept			
2				s5			3	4
3		s6	s7					
4		58						
5	s2	r5/r6	r5			9		
6		r1	r1		r1			
7		r3	r3		r3			
8		r2	r2		r2			
9		r4	r4					

Q.3. (b) Discuss ambiguity of a CFG.How Ambiguity can be removed from the grammar ? State suitable example. (5)

Ans. If a context free grammar G has more than one derivation tree for string $w \in L(G)$, it is called an **ambiguous grammar. There exist multiple right-most or left-most derivations for some string generated from that grammar.**

Problem

To check whether the grammar G with production rules–

$X \rightarrow X+X \mid X^*X \mid X^* \mid a$ is ambiguous or not.

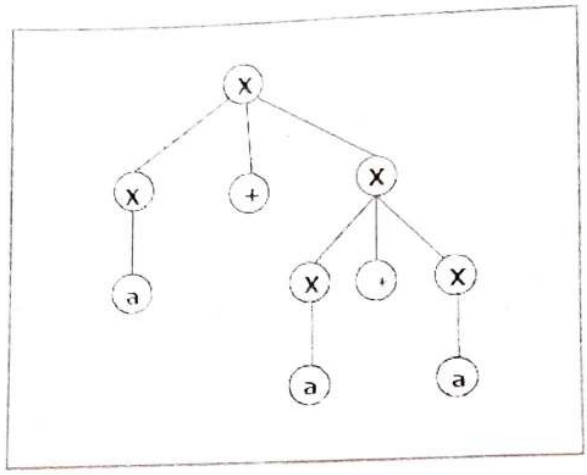
Solution

Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

Derivation 1 –

$X \rightarrow X+X \rightarrow a+X \rightarrow a+X*X \rightarrow a+a*X \rightarrow a+a*a$

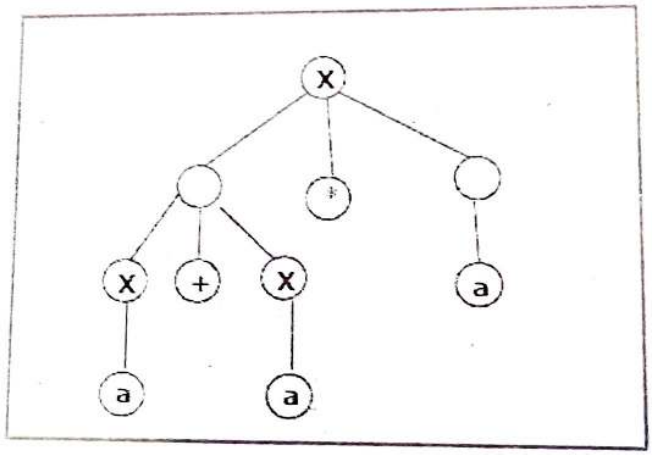
Parse tree 1 –



Derivation 2–

$X \bullet \rightarrow X*X \bullet \rightarrow X+X*X \bullet \rightarrow a+X*X \bullet \rightarrow a+a*X \rightarrow a+a*a$

Parse tree 2 –



As there are two parse trees for a single string "a+a*a", the grammar G is ambiguous.

Removal of Ambiguity:

1. Precedence of Operators is not respected.

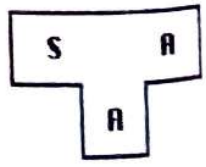
2.Sequence of identical operators can group either from left or from right .We would see two different parse tree for the above expression. Since addition is associative, it doesn't matter w

Q.4.Write short notes on following:

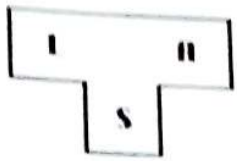
(a) Bootstrapping and Cross Compiler **(2.5)**

Ans: A compiler is characterized by three languages:

1. Source Language
2. Target Language
3. Implementation Language



Notation ${}_{S}C_{I}^{T}$ represents a compiler for Source S, Target T, implemented in I. The T-diagram shown above is also used to depict the same compiler.



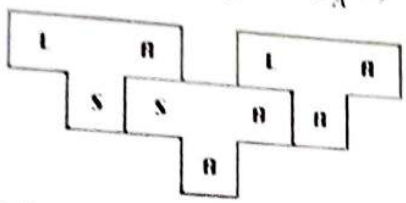
To create a new language, L, for machine A:

1.Create ${}_{S}C_{A}^{A}$, a compiler for a subset, S, of the desired language, L, using language A which runs on machine A. (Language A may be assembly language.)

2.Create ${}_{L}C_{S}^{A}$, a compiler for language L written in a subset of L.

3.Compile ${}_{L}C_{S}^{A}$ using ${}_{S}C_{A}^{A}$ to obtain ${}_{L}C_{A}^{A}$, a compiler for language L, which runs on machine A and produces code for machine A.

$${}_{L}C_{S}^{A} \rightarrow {}_{S}C_{A}^{A} \rightarrow {}_{L}C_{A}^{A}$$



The process illustrated by the T-diagrams is called *bootstrapping* and can be summarized by the equation:

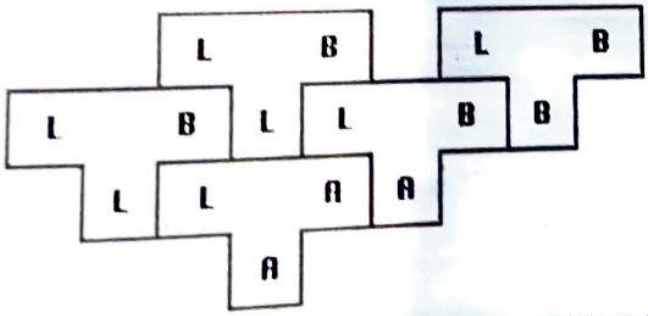
$$L_S\text{^} + S_A\text{^} = L_A\text{^}$$

To produce a compiler for a different machine B:

1.Convert ${}_{L}C_{S}^{A}$ into ${}_{L}C_{L}^{B}$ (by hand, if necessary). Recall that language S is a subset of language L.

2.Compile ${}_{L}C_{L}^{B}$ to produce ${}_{L}C_{A}^{B}$, a *cross-compiler* for L which runs on machine A and produces code for machine B.

3.Compile ${}_{L}C_{L}^{B}$ with the cross-compiler to produce ${}_{L}C_{B}^{B}$, a compiler for language L which runs on machine B.



Cross Compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler.

A cross compiler is necessary to compile for multiple platforms from one machine. A platform could be infeasible for a compiler to run on, such as for the microcontroller of an embedded system because those systems contain no operating system.

In paravirtualizationone machine runs many operating systems, and a cross compiler could generate an executable for each of them from one main source.

Cross compilers are not to be confused with source-to-source compilers. A cross compiler is for cross-platform software development of binary code, while a source-to-source compiler translates from one programming language to another in text code. Both are programming tools.

Uses of cross compilers

The fundamental use of a cross compiler is to separate the build environment from the target environment. This is useful in a number of situations:

- Embedded computers where a device has extremely limited resources. For example, a microwave oven will have an extremely small computer to read its touchpad and door sensor, provide output to a digital display and speaker, and to control the machinery for cooking food. This computer will not be powerful enough to run a compiler, a file system, or a development environment. Since debugging and testing may also require more resources than are available on an embedded system, cross-compilation can be less involved and less prone to errors than native compilation.
- Compiling for multiple machines. For example, a company may wish to support several different versions of an operating system or to support several different operating systems. By using a cross compiler, a single build environment can be set up to compile for each of these targets.
- Compiling on a server farm. Similar to compiling for multiple machines, a complicated build that involves many compile operations can be executed across any machine that is free, regardless of its underlying hardware or the operating system version that it is running.
- Bootstrapping to a new platform. When developing software for a new platform or the emulator of a future platform, one uses a cross compiler to compile necessary tools such as the operating system and a native compiler.
- Compiling native code for emulators for older now-obsolete platforms like the Commodore 64 or Apple II by enthusiasts who use cross compilers that run on a current platform (such as Aztec C's MS-DOS 6502 cross compilers running under Windows XP).

Use of virtual machines (such as Java's JVM) resolves some of the reasons for which cross compilers were developed. The virtual machine paradigm allows the same compiler output to be used across multiple target systems, although this is not always ideal because virtual machines are often slower and the compiled program can only be run on computers with that virtual machine.

Q.4. (b) Difference between compiler and interpreter (2.5)

Ans.

Interpreter	Compiler
1. Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
2. It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
3. No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
4. Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
5. Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

Q.4. (c) Handle and Handle Prunning **(2.5)**

Ans: *Handles:*

o A *handle* of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation.

- ***Precise definition of a handle:***

o A *handle* of a right-sentential form γ is a production $A \to \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

o i.e., if $S \ \alpha Aw \ \alpha\beta w$, then $A \to \beta$ in the position following α is a handle of $\alpha\beta w$.

o The string w to the right of the handle contains only terminal symbols.

o In the example above, abbcde is a right sentential form whose handle is $A \to b$ at position 2. Likewise, aAbcde is a right sentential form whose handle is $A \to Abc$ at position 2.

- ***Handle Pruning:***
- A rightmost derivation in reverse can be obtained by *handle pruning*.
- i.e., start with a string of terminals w that is to parse. If w is a sentence of the grammar at hand, then$w = \to\gamma_n$, where $\to\gamma_n$ is the nth right sentential form of some as yet unknown rightmost derivation.
- $$S = \gamma_0 \ \gamma_1 \ \gamma_2 \ \cdots \gamma_{n-1} \ \gamma_n = w.$$

Example for right sentential form and handle for grammar

$E \to E + E$

$E \to E * E$

$E \to (E)$

$E \to id$

Right Sentental Form	Handle	Reduction Production
id1 + id2 * id3	id1	$E \to id$
E + id2 * id3	id2	$E \to id$
E + E * id3	id3	$E \to id$
E + E • E	E * E	$E \to E * E$
E + E	E + E	$E \to E + E$
E		

(2.5)

Q.4. (d) LEX Tool

Ans: Lex is a computer program that generates lexical analyzers ("scanners" or "lexers").

Lex is commonly used with the yacc parser generator. Lex, originally written by Mike Lesk and Eric Schmidt and described in 1975, is the standard lexical analyzer generator on many Unix systems, and an equivalent tool is specified as part of the POSIX standard.

Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language

Open source

Though originally distributed as proprietary software, some versions of Lex are now open source. Open source versions of Lex, based on the original AT&T code are now distributed as open source systems such as OpenSolaris. One popular open

END TERM EXAMINATION [MAY-JUNE 2017]
SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS–302]

Time : 3 Hrs. M.M. : 75

Note: Attempt any five questions include Q. No 1 which is compulsory. Select one question from each unit.

Q.1. Attempt any five parts:

Q.1. (a) Explain the process of Bootstrapping in compiler design with example. (5)

Ans. Refer Q. No. 4 (a) First Term Examination 2017.

Q.1. (b) Differentiate between SDD and SDT with Example. (5)

Ans. SYNTAX-DIRECTED TRANSLATION(SDT) A formalism for specifying translations for programming language constructs. (attributes of a construct: type, string, location, etc)

- Syntax directed definition(SDD) for the translation of constructs
- Syntax directed translation scheme(SDTS) for specifying translation Postfix notation for an expression E
- If E is a variable or constant, then the postfix nation for E is E itself (E.t ≡ E).
- if E is an expression of the form E1 op E2 where op is a binary operator o E1' is the postfix of E1, o E2' is the postfix of E2 o then E1' E2' op is the postfix for E1 op E2
- if E is (E1), and E1' is a postfix then E1' is the postfix for E

eg) 9 - 5 + 2 ⇒ 9 5 - 2 + 9 - (5 + 2) ⇒ 9 5 2 + -

Syntax-Directed Definition(SDD) for translation

- SDD is a set of semantic rules predefined for each productions respectively for translation.
- A translation is an input-output mapping procedure for translation of an input X, o construct a parse tree for X. o synthesize attributes over the parse tree.
- Suppose a node n in parse tree is labeled by X and X.a denotes the value of attribute a of X at that node.
- compute X's attributes X.a using the semantic rules associated with X.

Q.1. (c) What is Left Recursion and Left Factoring ? Explain each with Example. (5)

Ans. Left Recursion

- A grammar is left recursive iff it contains a nonterminal A, such that A→+ Aα, where is any string.

Grammar {S → Sα | c} is left recursive because of S ⇒Sα

Grammar {S →Aα, A → Sb | c} is also left recursive because of S⇒Aα⇒Sbα

- If a grammar is left recursive, you cannot build a predictive top down parser for it.

(1) If a parser is trying to match S & S →Sα, it has no idea how many times S must be applied (2) Given a left recursive grammar, it is always possible to find another grammar that generates the same language and is not left recursive. 3) The resulting grammar might or might not be suitable for RDP

source version of Lex, called flex, or the "fast lexical analyzer", is not derived from proprietary coding.

Structure of a Lex file

The structure of a Lex file is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows

- The **definition** section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
- The **rules** section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.
- The **C code** section contains C statements and functions that are copied to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

Example of a Lex file

The following is an example Lex file for the flex version of Lex. It recognizes strings of numbers (positive integers) in the input, and simply prints them out.

```
/*** Definition section ***/
%{
/* C code to be copied verbatim */
#include <stdio.h>
%}
/* This tells flex to read only one input file */
%option noyywrap
%%
/*** Rules section ***/
/* [0-9]+ matches a string of one or more digits */
[0-9]+ {
/* yytext is a string containing the matched text. */
printf("Saw an integer: %s\n", yytext);
}
. | \n { /* Ignore all other characters. */ }
%%
/*** C Code section ***/
int main(void)
{
/* Call the lexer, then quit. */
yylex();
return 0;
}
```

If this input is given to flex, it will be converted into a C file, lex.yy.c. This can be compiled into an executable which matches and outputs strings of integers. For example, given the input:

abc123z. ! &*2gj6

the program will print:

Saw an integer: 123
Saw an integer: 2
Saw an integer: 6

• After this, if we need left factoring, it is not suitable for RDP.

• Right recursion: Special care/Harder than left recursion/SDT can handle.

S $\rightarrow$ Aa | b

A $\rightarrow$ **Ac/Sd/e** As we don't have immediate left recursion,

Step(1) Rename S,A as A_1,A_2 respectively

$A_1 \rightarrow A_2a \mid b$

$A_2 \rightarrow A_2c \mid A1\,d \mid e$

Step(2) Substitute value A_1 of statement (1) in R.H.S of statement (2)

$A_1 \rightarrow A_2a \mid b$

$A_2 \rightarrow A_2c \mid A_2a\,d \mid bd \mid e$

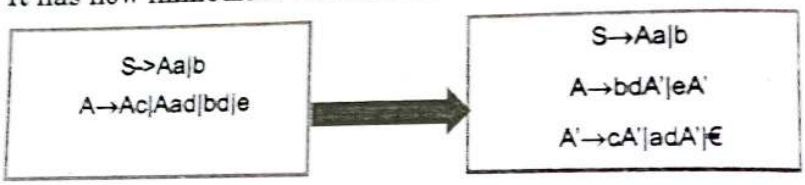
Step(3)

Again substitute $A_1 = S, A_2 = A$

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Aad \mid bd \mid e$

Step (4) It has now immediate left recursion



Eliminating Left Recursion Let G be S → S A | A Note that a top-down parser cannot parse the grammar G, regardless of the order the productions are tried. ⇒ The productions generate strings of form AA..A

⇒ They can be replaced by S →A S' and S →A S' | ε

Left Factoring

• If a grammar contains two productions of form S → aβ and S →aα it is not suitable for top down parsing without backtracking. Troubles of this form can sometimes be removed from the grammar by a technique called the left factoring.

• In the left factoring, we replace { S→ aα, S→ aβ } by { S → aS′, S′ → α, S′ → β } or S → a(α | β) (Hopefully α and β start with different symbols)

• left factoring for G { S → aSb | c | ab } S → aS' | c cf. S(=aSb | ab | c = a (Sb | c) → a S' | c S'→Sb | b

Q.1. (d) What is Back Patching? Explain with Example. (5

Ans. BACKPATCHING

The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations. The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. Hence, a series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels backpatching.

To manipulate lists of labels, we use three functions :

1. makelist(i) creates a new list containing only i, an index into the array of quadruples; makelist returns a pointer to the list it has made.

2. merge(p1,p2) concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenated list.

3. backpatch(p,i) inserts i as the target label for each of the statements on the list pointed to by p.

Boolean Expressions:

We now construct a translation scheme suitable for producing quadruples for boolean expressions during bottom-up parsing. The grammar we use is the following:

(1) E E1 or M E2
(2) | E1 and M E2
(3) | not E1
(4) | (E1)
(5) | id1 relop id2
(6) | true
(7) | false
(8) M → ε

Synthesized attributes truelist and falselist of nonterminal E are used to generate jumping code for boolean expressions. Incomplete jumps with unfilled labels are placed on lists pointed to by E.truelist and E.falselist.

Q.1. (e) What is the Process of identifying basic blocks in code optimization phase?

Ans. OPTIMIZATION OF BASIC BLOCKS (5)

There are two types of basic block optimizations. They are :

- Structure-Preserving Transformations
- Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- Common sub-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements.

Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

a: =b+c
b: =a-d
c: =b+c
d: =a-d

The 2nd and 4th statements compute the same expression: b+c and a-d

Basic block can be transformed to

a: = b+c
b: = a-d
c: = a
d: = b

Dead code elimination:

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to

remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

Renaming of temporary variables:

- A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.
- In this we can transform a basic block to its equivalent block called normal-form block.

Interchange of two independent adjacent statements:

Two statements

t1 :=b+c

t2 :=x+y

can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

Algebraic Transformations:

Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.

Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.

he relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.

Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

a :=b+c e :=c+d+b

the following intermediate code may be generated:

a :=b+c t :=c+d e :=t+b

Example:

x :=x+0 can be removed

x :=y**2 can be replaced by a cheaper statement x:=y*y

The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

Q.1. (f) Differentiate between top-down and bottom-up parsers with example. (5)

Ans. Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

- Recursive descent parsing : It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.
- Backtracking : It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Bottom-up Parsing

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

Example:

Input string : a + b * c

Production rules:

$S \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow E * T$

$E \rightarrow T$

$T \rightarrow id$

Let us start bottom-up parsing

a + b * c

Read the input and check if any production matches with the input:

a + b * c

T + b * c

E + b * c

E + T * c

E * c

E * T

E

S

Q.1. (g) Write a SDT for converting infix expression to post fix expression by taking suitable example. **(5)**

Ans. If you start with an infix expression, the following algorithm will give you the equivalent postfix expression.

- Variables and constants are left alone.
- E op F becomes E' F' op, where E' and F' are the postfix of E and F respectively.
- (E) becomes E', where E' is the postfix of E.

1+2/3-4*5

1. Start with 1+2/3-4*5
2. Parenthesize (using standard precedence) to get (1+(2/3))-(4*5)
3. Apply the above rules to calculate P{(1+(2/3))-(4*5)}, where P{X} means "convert the infix expression X to postfix".

A. P{(1+(2/3))-(4*5)}

B. P{(1+(2/3))} P{(4*5)} -

C. P{1+(2/3)} P{4*5} -

D. P{1} P{2/3} + P{4} P{5} * -

E. 1 P{2} P{3} / + 4 5 * -

F. 1 2 3 / + 4 5 * -

UNIT–I

Q.2. (a) For the grammar given below:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon,$$
$$F \rightarrow (E) \mid ID$$

construct the LL(1) parsing table

Ans. First( ) :

FIRST(E) = { (, id}
FIRST(E') ={+ , ε }
FIRST(T) = { (, id}
FIRST(T') = {*, ε }
FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }
FOLLOW(E') = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T') = { +, $,) }
FOLLOW(F) = {+, * , $,) }

Predictive parsing table:

NON-TERMINAL	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → + TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Stack implementation:

Stack	Input	Output
$E	id + id*idS	
$E'T	id+id*id$	E → TE'
$ET'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id$	
$E'	+id*id$	T' → ε
$E'T+	+id*id$	E' → + TE'
$E'T	id*id$	
$E'T'F	id*id$	T → FT'
$E'T'id	id*id$	F → id
$E'T'	*id$	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

Q.2. (b) Check Whether the following grammar is LL(1) or not (5)

i. $S \rightarrow A/a, A \rightarrow a$

ii. $S \rightarrow aSA/\varepsilon, A \rightarrow c/\varepsilon$

Ans. acc to the rule:

(1) First(A) → First(a) = {a}

(2) First(S) → First (ε) = ε

this grammar doesnot follow the rules that's why it is not LL(1)

Q.3. (a) What do you mean by handle? Check whether the grammar (5)

$E \rightarrow E + T/T$

$T \rightarrow$ a or (id)is LR(0) or not

Ans. *Handles:*

o A *handle* of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation.

- ***Precise definition of a handle:***

o A *handle* of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

o i.e., if $S \Rightarrow \alpha Aw \Rightarrow \alpha\beta w$, then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta w$.

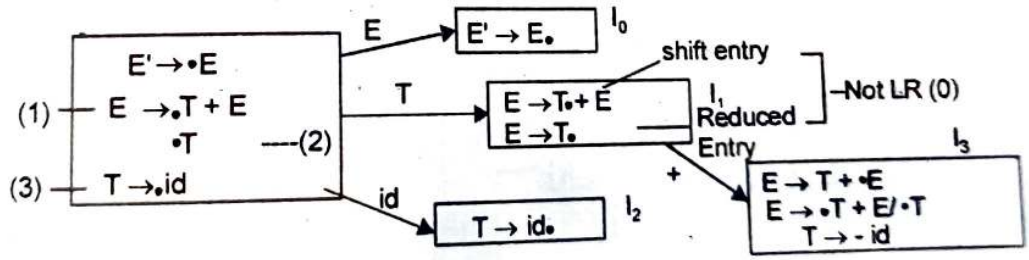
o The string w to the right of the handle contains only terminal symbols.

o In the example above, abbcde is a right sentential form whose handle is $A \rightarrow b$ at position 2. Likewise, aAbcde is a right sentential form whose handle is $A \rightarrow Abc$ at position 2. the augmented grammar is:

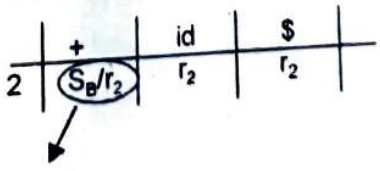
$E' \rightarrow E$

$E \rightarrow T+E/T$

$T \rightarrow a$



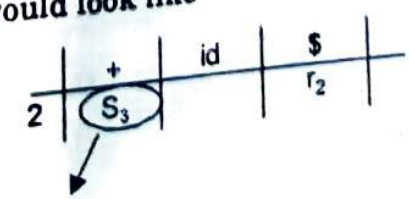
The LR(0) parsing table would look like



SR conflict → Not LR(0)

But for SLR E → T. reduced entry will go only in follow (E) = $

So SLR parsing table would look like



dotnotes.xyz

LR (1) Parsing Table

State	Action				goto			
	a	b	c	d	$	S	A	B
0		s3		s5		1	2	4
1					accept			
2	s6							
3				s9			7	8
4			s10					
5	r5		r6					
6					r1			
7			s11					
8	s12							
9	r6		r5					
10					r3			
11					r2			
12					r4			

So, the LR (1) Parsing table has no multiple entries ∴ Grammar is LR (1).

UNIT–2

Q.4. (a) Write an SDT to count the number of binary digits in abinary number **(5)**

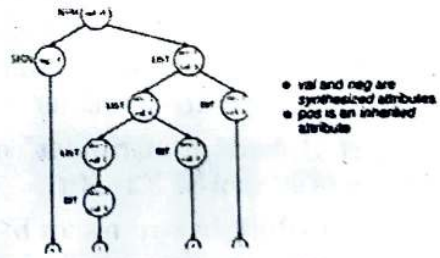
Ans:

Example: Evaluate signed binary numbers

PRODUCTION	SEMANTIC RULES
NUM → SIGN LIST	LIST.pos = 0 if SIGN.neg NUM.val = -LIST.val else NUM.val = LIST.val
SIGN → +	SIGN.neg = false
SIGN → -	SIGN.neg = true
LIST → BIT	BIT.pos = LIST.pos LIST.val = BIT.val
LIST → LIST, BIT	$LIST_1$.pos = LIST.pos + 1 BIT.pos = LIST.pos LIST.val = $LIST_1$.val + BIT.val
BIT → 0	BIT.val = 0
BIT → 1	BIT.val = $2^{BIT.pos}$

Example (continued)

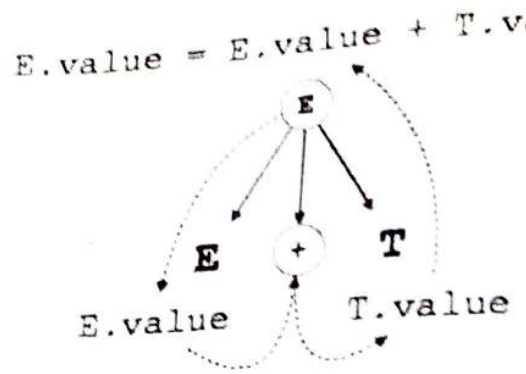
The attributed parse tree for –101:



Q.4. (b) Differentiate between S-attributed and L-attributedSDT's.Write the steps to create the SDT for any problem and write SDT for converting any number from binary to decimal. **(7.5)**

Ans. S-Attributed Grammars are a class of attribute grammars characterized by having no inherited attributes, but only synthesized attributes. Inherited attributes, which must be passed down from parent nodes to children nodes of the abstract syntax tree during the semantic analysis of the parsing process, are a problem for bottom-up parsing because in bottom-up parsing, the parent nodes of the abstract syntax tree are created *after* creation of all of their children. Attribute evaluation in S-attributed grammars can be incorporated conveniently in both top-down parsing and bottom-up parsing.

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed grammars are a special type of attribute grammars. They allow the attributes to be evaluated in one depth-first left-to-right traversal of the abstract syntax tree. As a result, attribute evaluation in L-attributed grammars can be incorporated conveniently in top-down parsing.

A syntax-directed definition is L-attributed if each inherited attribute of Xj on the right side of

$$A \rightarrow X1\ X2\ ..Xn$$

depends only on

1. The attributes of the symbols X1, X2, ..., Xj-1
2. The inherited attributes of A

Every S-attributed syntax-directed definition is also L-attributed.

Implementing L-attributed definitions in Bottom-Up parsers requires rewriting attributed definitions into translation schemes.

Many programming languages are L-attributed. Special types of compilers, the narrow compilers, are based on some form of L-attributed grammar. These are a strict superset of S-attributed grammars. Used for code synthesis.

Either "Inherited attributes" or "synthesized attributes" associated with the occurrence of symbol X1,X2---Xn

Q.5. (a) what do you mean by three address code? Explain how the three address code is represented visquadruples,triples and indirect triples with examples.

Ans: Three-Address Code: **(7.5)**

Three-address code is a sequence of statements of the general form

x : = y op z

where x, y and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on booleanvalued data. Thus a source language expression like x+ y*z might be translated into a sequence

t1 : = y * z t2 : = x + t1

where t1 and t2are compiler-generated temporary names.

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the

operands. Three such representations are:

Quadruples

Triples

Indirect triples

Quadruples:

→ A quadruple is a record structure with four fields, which are, op, arg1, arg2 and result.

→ The op field contains an internal code for the operator. The three-address statement x : = y op z is represented by placing y in arg1, z in arg2 and x in result.

→ The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples:

→ To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.

→ If we do so, three-address statements can be represented by records with only three fields: op, arg1 and arg2.

→ The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure (for temporary values).

→ Since three fields are used, this intermediate code format is known as triples.

Indirect Triples:

Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples

Q.5.(b) Write the three address code for: **(5)**

(I) While (a < 5) do a : b + 2

(II) – a (a + b)* (c + d) + (a + b + c)

Ans. (i) While (a<5) do a:b +2

1. if a<5
2. T1 = b+2
3. a = T1

(ii) –a (a+b)*(c+d) +(a+b+c)

Ans: t1 = a+b

t2 = -uminus(a)

t3 = c+d

t4 = t1*t3

t5 = t1+c

t6 = t3 +t5

UNIT-III

Q.6. (a) What do you mean by symbol table? Write an example that shows how different phases of compiler interact with symbol table. **(6)**

Ans: Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

26 2017

A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

<symbolname, type, attribute>

Q.6. (b) How the data is stored in symbol table for block and non block structured languages? (6.5)

Ans. Symbol table organization for non-blocked structured language:

By a non blocked structure language, we mean a language in which each separately compiled unit is a single module that has no submodules. All variables declared in a module are known throughout the module.

There are four modules

- Unordered
- Ordered
- Tree
- Hash

1. Unordered

The simplest method of organizing a symbol table is to add the attribute entries to the table in the order in which the variables are declared. So there is no particular ordering.

- Here the insertion is very easy as no comparision are required.
- The searching is more difficult hence it is very time consuming.
- For delete operation, on the average, a search length of (n+1)/2 is required, ing there a n records.

Time to insert a key → 1

Unsuccessful search the time wanted is (n+1)

unwanted table organization should be used only if the expected size of the table is small, since the average time for insertion and deletion is directly proportional to the table size.

The algorithm is given for selecting a data whose key is known to us.

```
procedure select(table, key, data value, found)
  var i unteger;
  begin
        with table do
            begin
                entry[n+1].key = key;
                i=1;
                while(entry[i].key<>key
                    i=i+1;
                if i<=n then
```

```
                    begin
                    data value=entry[i].value;
                    found=TRUE;
                    end
                else
                    found=FALSE;
            end
    end
```

So the complexity is O((n+1)/2). .....

Now let's see various aspects: (Actual use)

(i) Fixed as variable length entries.

In fixed , the number of entries possible in a table are fix where as in variable length it is changed.

(ii) Size of key(in byte).

(iii) Method of access

(iv) How frequent the insertion and deletion.

Language specific consideration:

(i) BASIC

Variables are stored as,

<letter> as <letter><digit>

So with this format at the most 286 entries are possible. To know the address of a given key, we have formula as follows:

<Address> = <key> - 65 OR

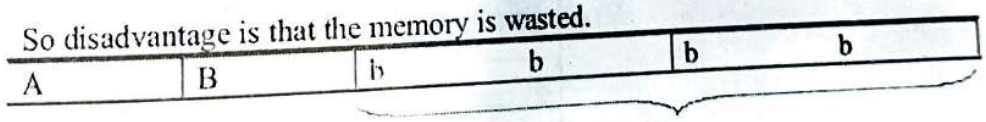
<Address> = (< key – letter> – 65) * 10 + <digit> - 48

(ii) FORTRAN

In this language the fixed sized length of variable is used.

So disadvantage is that the memory is wasted.

So disadvantage is that the memory is wasted.

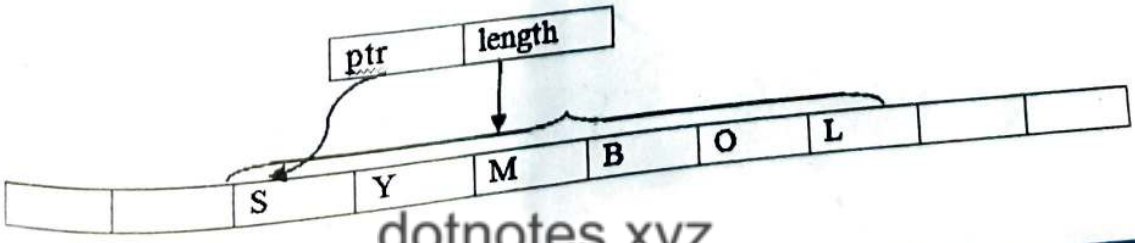


Padding (This much bytes are wasted)

(iii) C or PASCAL

In such high-level languages, the variable length key – strange are used.

With this type of key strange, we get the better memory utilization but the method is not fast.



2. Ordered

In this method, the table is maintained is sorted form based on the variable name. In such circumstances an insertion must be accomplished by a lookup procedure which determines where in the symbol table the variable attribute should be placed. The actual insertion of new entry may generate. Some additional overhead primarily because other entries may have to be moved to get the position of insertion.

For searching a particular key, we apply Binary-Search Technique. Suppose (K_1, V_1) .... (K_{mid}, V_{mid}) .... (K_n, V_n) are the entries in the table.

Here mid= n div 2

Algorithm is described below

```
Find(low, high)
While low < high do
begin
    mid = (low+high) div 2;
    if k < entry[mid].key then
                        high = mid;
                    Find(low,high);
    else
                        low = mid + 1;
                    Find(low,high);
end
```

So the key for which we are sending is placed in high or low variable.

The time complexity of this algorithm is O(log n).

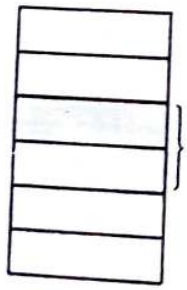
Methods of Sorting:

i. Array

We short the entries in the table in some particular .....with arrays.

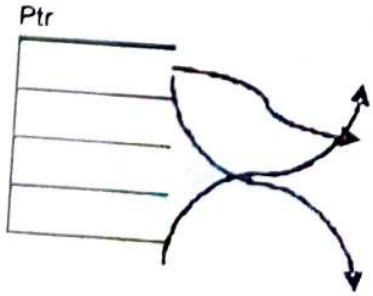
With arrays, the searching of particular entry is very fast. But insertion is time consuming.

For inserting particular entry, first we have to find its position to locate it. And all the entries below it are shifted down.



ii. Index

With this method, the insertion is easy.

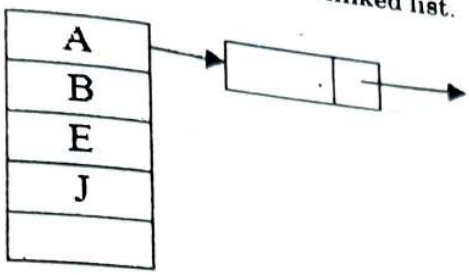


K	V

Here only ptr field is manipulated. We have to do nothing with table. So, it is easier for insertion.

iii. Linked List

In this approach, we combined the array and linked list.



Here, array is for searching and linked list is used for insertion and deletion. Here there is no actual limit of number of entries in the table.

To search a key, starting symbol can be found by comparison and then entries can be counted to find the exact match.

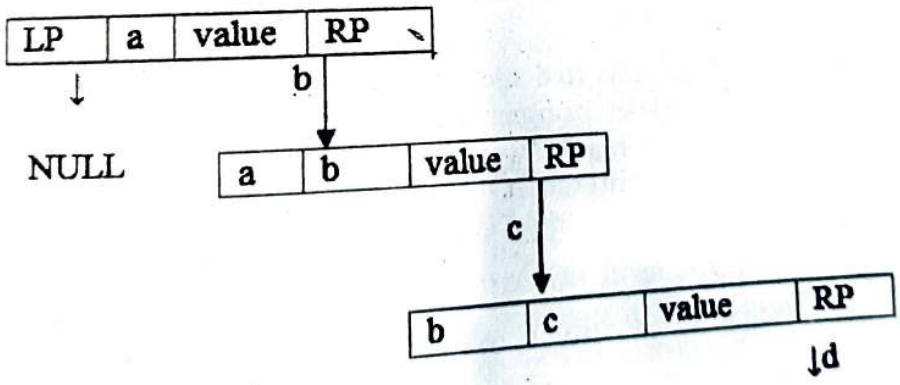
3. Tree

In a binary – tree structured symbol table, each node have the following format:

Left ptr	Key	Value	Right ptr

Here two new fields are present in the record structure. Thus two fields are left pointer and right pointer. Access to the tree is gained through the ... node. A search proceeds down the structural links of the tree until the desired node is found as a NULL link field is encounted.

Let's take an example of storing a string abcd in this format.



Here in case of balanced binary tree the time complexity of a searching a node among the n node is given by $O(\log_2 n)$

4. Hash

A hashing function or key to address transformation is defined as a mapping H: K→A. That is, a hashing function H takes as its argument a variable named and produces a table address at which the set of attributes for that variables are stored. With this method the search time is essentially independent of the number of records in the table.

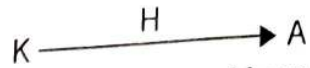


table space address space

Let n be the number of entries in the table we define loading factor,

load factor = no of entries(n) / total address space (| A |)

If load factor is high, it is difficult to manage the table.

Now in practical we have

$$K >> |A|$$

So if we assign more than one key to one address, there is a problem of collision.

Pre conditioning:-

(1) Division Method:-

(2) Mid-square method:-

(3) Folding Method:-

(i) Length-dependent method:-

(a) Open Addressing:

(b) Chaining:

Symbol-Table Organization for Blocked Structured Language:-

(i) Stack symbol tables

(ii) Stack implemented tree structural tables

(iii) Stack – Implemented Hash-structured Symbol Table:

Q.7. (a) What are different types of errors that occur during lexical,syntactic and semantic phase. (6)

Ans: A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for ors but errors may be encountered at various stages of the compilation process. A am may have the following kinds of errors at various stages:

- **Lexical** : name of some identifier typed incorrectly
- **Syntactical** : missing semicolon or unbalanced parenthesis
- **Semantical** : incompatible value assignment
- **Logical** : code not reachable, infinite loop

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

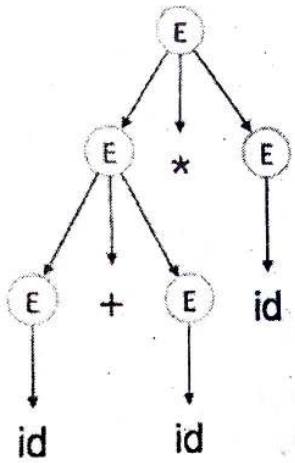
Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

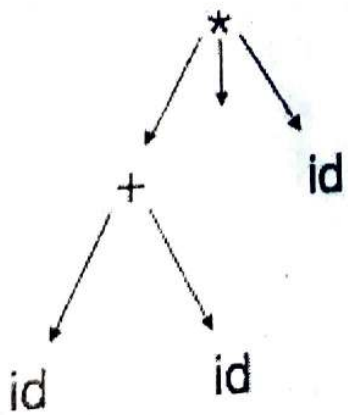
The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Abstract Syntax Trees

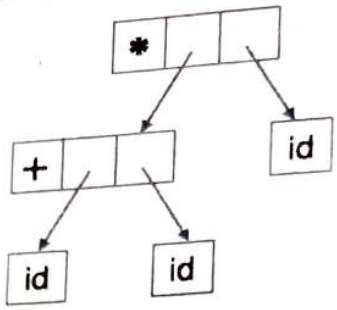
Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed. Take the following parse tree as an example:



If watched closely, we find most of the leaf nodes are single child to their parent nodes. This information can be eliminated before feeding it to the next phase. By hiding extra information, we can obtain a tree as shown below:



Abstract tree can be represented as:



ASTs are important data structures in a compiler with least unnecessary information. ASTs are more compact than a parse tree and can be easily used by a compiler.

Q.7. (b) What are the storage allocation strategies in the runtime environment of compiler? (6.5)

Ans: A program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.

By runtime, we mean a program in execution. Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.

Runtime support system is a package, mostly generated with the executable program itself and facilitates the process communication between the process and the runtime environment. It takes care of memory allocation and de-allocation while the program is being executed.

Activation Trees

A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.

The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries: Stores temporary and intermediate values of an expression.

Local Data: Stores local data of the called procedure.

Machine Status: Stores machine status such as Registers, Program Counter etc., before the procedure is called.

Control Link: Stores the address of activation record of the caller procedure.

Access Link: Stores the information of data which is outside the local scope.

Actual Parameters: Stores actual parameters, i.e., parameters which are used to send input to the called procedure.

Return Value: Stores return values.

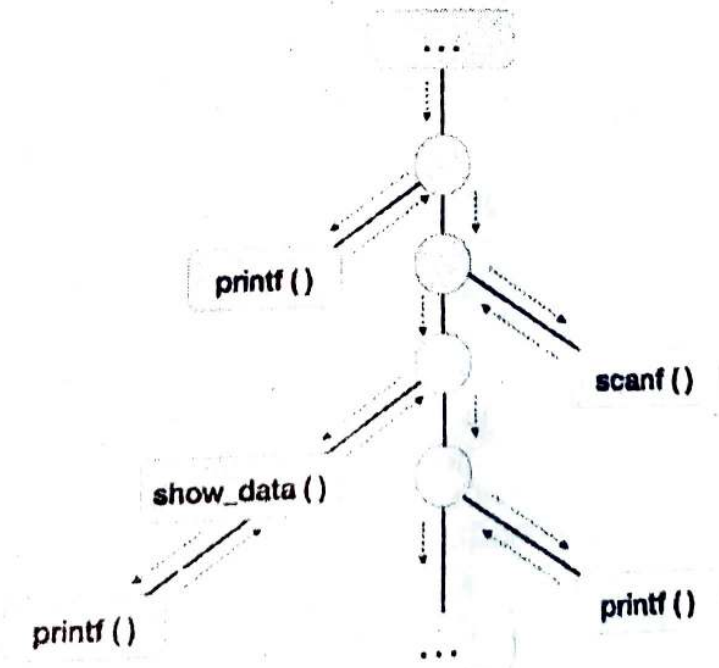
Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.

We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the **activation tree.**

To understand this concept, we take a piece of code as an example:

```
...
printf("EnterYourName:");
scanf("%s", username);
show_data(username);
printf("Press any key to continue..");
...
intshow_data(char*user)
{
printf("Your name is%s", username);
return0;
}
...
```

Below is the activation tree of the code given.



Now we understand that procedures are executed in depth-first manner, thus stack allocation is the best suitable form of storage for procedure activations.

Storage Allocation

Runtime environment manages runtime memory requirements for the following entities:

- **Code** : It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.
- **Procedures** : Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- **Variables** : Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Static Allocation

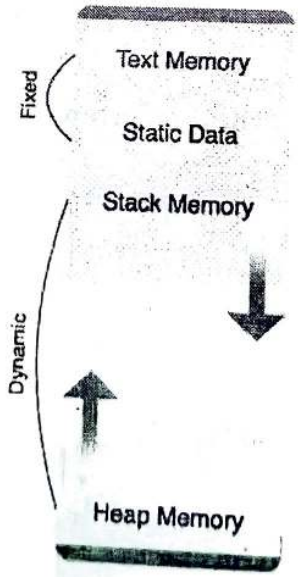
In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

Stack Allocation

Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

Heap Allocation

Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.



Except statically allocated memory area, both stack and heap memory can grow hrink dynamically and unexpectedly. Therefore, they cannot be provided with a amount of memory in the system.

As shown in the image above, the text part of the code is allocated a fixed amount memory. Stack and heap memory are arranged at the extremes of total memory llocated to the program. Both shrink and grow against each other.

UNIT-4

Q.8. (a) What do you mean by the term code optimization? What do you understand by the term leader? Write algorithm to identify out the basic blocks. (6)

Ans. Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Basic Blocks: A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

The following sequence of three-address statements forms a basic block:

```
t1 : = a * a
t2 : = a * b
t3 : = 2 * t2
t4 : = t1 + t3
t5 : = b * b
t6 : = t4 + t5
```

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:We first determine the set of leaders, the first statements of basic blocks. The rules we use are of the following:

The first statement is a leader.

Any statement that is the target of a conditional or unconditional goto is a leader.Any statement that immediately follows a goto or conditional goto statement is a leader.

For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Consider the following source code for dot product of two vectors a and b of length 20 begin

```
prod :=0;
i:=1;
do begin
```

```
prod :=prod+ a[i] * b[i];
i :=i+1;
end
while i <= 20
end
```

The three-address code for the above source program is given as :

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4* i
(4)   t2 := a[t1]     /*compute a[i] */
(5)   t3 := 4* i
(6)   t4 := b[t3]     /*compute b[i] */
(7)   t5 := t2*t4
(8)   t6 := prod+t5
(9)   prod := t6
(10)  t7 := i+1
(11)  i := t7
(12)  if i<=20 goto (3)
```

Q.8. (b) Identify the basic blocks in the following code and draw the DAG graph for the same: **(6.5)**

```
main()
{
int i = 0,n = 10;
int a[n];
while ( i< = (n-1))
{
a[i] = i*I;
i = i+1;
}
return;
}
```

Ans. Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.

For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is node(z). (Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not ch a node.

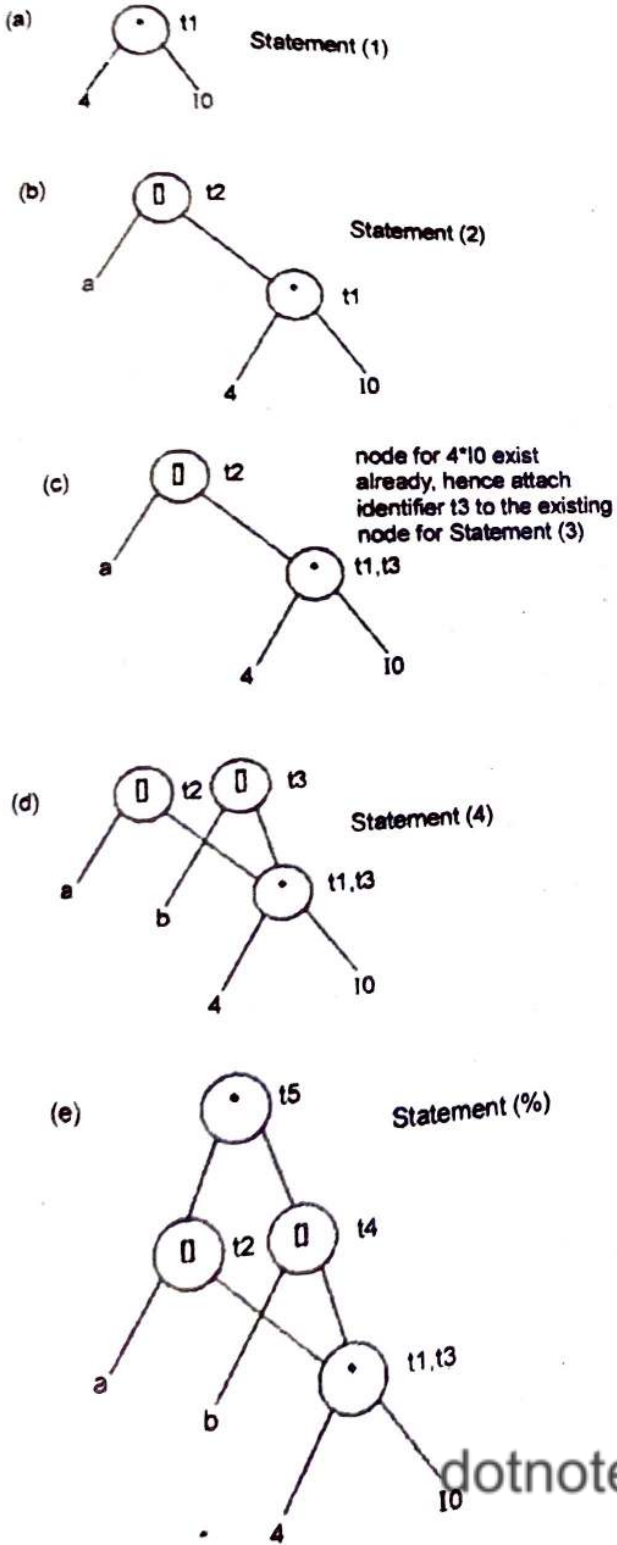
case(iii), node n will be node(y).

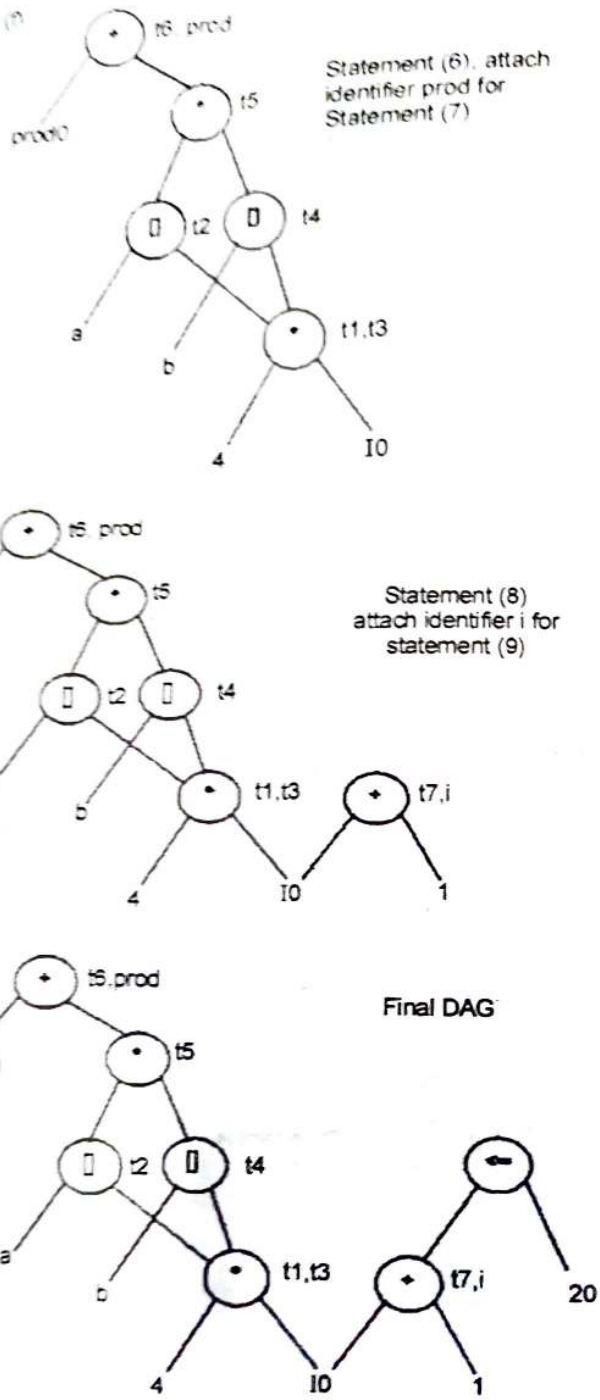
tep 3: Delete x from the list cf identifiers for node(x). Append x to the list of ched identifiers for the node n found in step 2 and set node(x) to n.

Example: Consider the block of three- address statements:

$t_1 := 4 * i$
$t_2 := a[t_1]$
$t_3 := 4 * i$
$t_4 := b[t_3]$
$t_5 := t_2 * t_4$
$t_6 := prod + t_5$
$prod := t_6$
$t_7 := i + 1$
$i := t_7$
if i<=20 goto (1)

Stages in DAG Construction





Q.8.(a) What do you mean by peephole optimization? Explain with example. **(6)**

Ans. Refer Q.1.(i) of End Term Examination 2016.

Q.9. (b) What are the issues that occurs during the code generation process? **(6.5)**

Ans. The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

1. Position of code generator
2. Source intermediate
3. Code program
4. Intermediate target program code

ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate representation can be : a. Linear representation such as postfix notation b. Three address representation such as quadruples c. Virtual machine representation such as stack machine code d. Graphical representations such as syntax trees and dags.

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

The output of the code generator is the target program. The output may be : a. Absolute machine language - It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language - It allows subprograms to be compiled separately.

c. Assembly language - Code generation is made easier.

3. Memory management:

Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.

It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.

Labels in three-address statements have to be converted to addresses of instructions. For example, j :goto i generates jump instruction as follows : if i < j, a backward jump instruction with target address equal to location of code for quadruple i is generated. if i > j, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to i are filled.

4. Instruction selection:

The instructions of target machine should be complete and uniform.

Instruction speeds and machine idioms are important factors when efficiency of target program is considered.

The quality of the generated code is determined by its speed and size.

The former statement can be translated into the latter statement as shown below:

5. Register allocation

→ Instructions involving register operands are shorter and faster than those involving operands in memory.

→ The use of registers is subdivided into two subproblems :

→ Register allocation – the set of variables that will reside in registers at a point in the program is selected.

→ Register assignment – the specific register that a variable will reside in is picked.

→ Certain machine requires even-odd register pairs for some operands and results. For example , consider the division instruction of the form : D x, y

where, x – dividend even register in even/odd register pair y – divisor even register holds the remainder odd register holds the quotient

6. Evaluation order : The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

FIRST TERM EXAMINATION [FEB. 2018]
SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS–302]

Time : 1.5 hrs. **M.M. : 30**

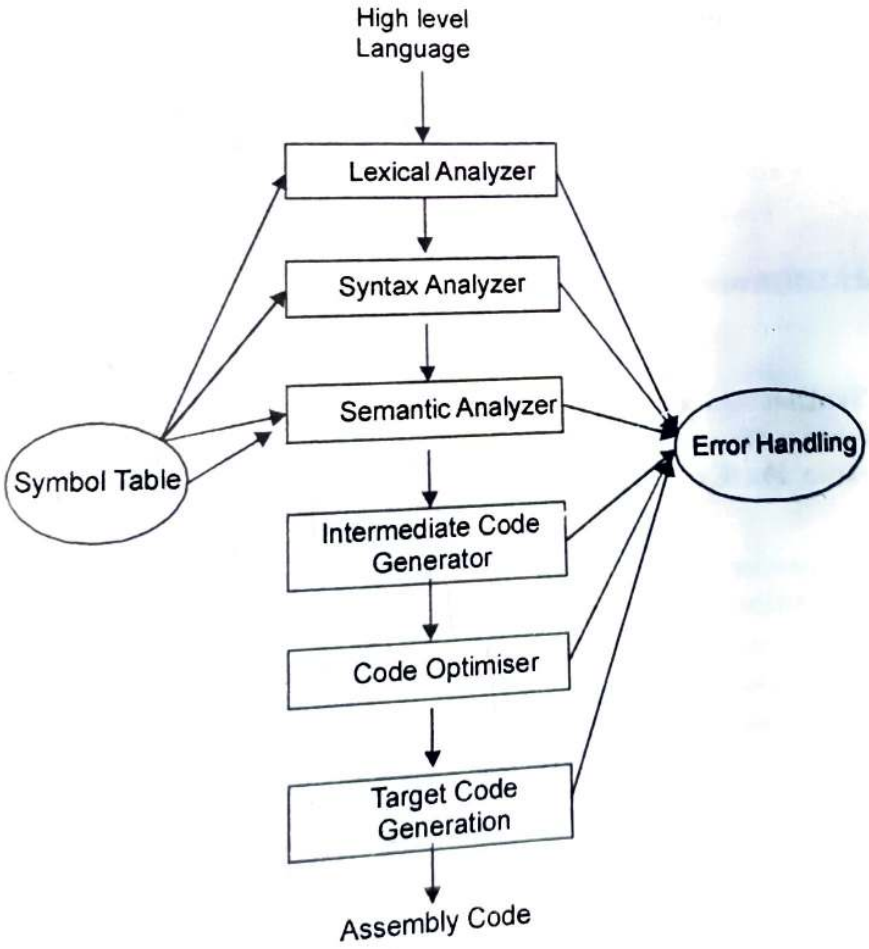
Note: Attempt Q. No. 1 which is compulsory and any two other questions from remaining questions. Each question carries 10 marks.

Q. 1. (a) List the various phases of compilation process. **(2)**

Ans. The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.

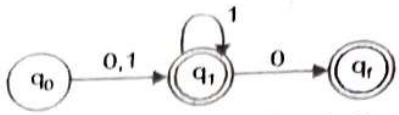
Following are the phases of compiler:

i. Lexical Analysis
ii. Syntax Analysis
iii. Semantic Analysis
iv. Intermediate Code Generation
v. Code Optimization
vi. Code Generation
vii. Symbol Table



Q. 1. (b) Generate a pattern matcher which can recognize following two patterns: o1* and 1*0. **(2)**

Ans.



Q. 1. (c) Improve the grammar by removing left recursion. S → AaB | bA → Aab | Aba | a B → a. **(2)**

Ans.

$S \rightarrow AaB/b$ is not left recursive

$B \rightarrow a$ is not left recursive

Only $A \rightarrow Aab \mid Aba \mid a$ is left recursive

$A \rightarrow Aab \mid a$ $\quad$ $A \rightarrow Aba \mid a$

$\boxed{\begin{matrix} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \in \end{matrix}}$ steps to remove left recursion

Here A is A	Here A is A
α is ab	α is ba
β is a	β is a
so, $A \rightarrow aA'$ ....(1)	so $A \rightarrow aA'$ ....(3)
$A' \rightarrow abA' \mid \in$ ....(2)	$A' \rightarrow baA' \mid \in$ ...(4)

(1) and (2) are same. So. instead of using 2 grammers

So, $\left.\begin{matrix} S \rightarrow AaB/b \\ A \rightarrow aA' \\ A' \rightarrow abA' \mid baA' \mid \in \\ B \rightarrow a \end{matrix}\right\}$ Final grammer after removing left recursion

Q. 1. (d) Differentiate between Synthesized attributes and inherited attributes. **(2)**

Ans. A Synthesized attribute is an attribute of the nonterminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. For Ex:

$$S \rightarrow ABC$$

If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

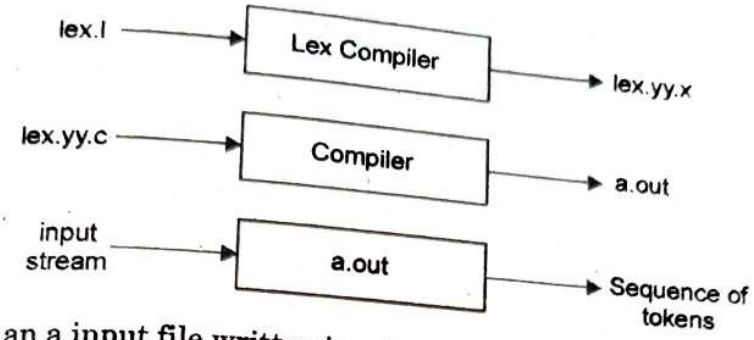
An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. An attribute that gets its values from the attributes attached to the parent or siblings of its non-terminal. For Ex:

$$S \rightarrow ABC$$

A can get values from S, B and C, B can take values from S, A, and C. Likewise, C take values from S, A, and B.

Q. 1. (e) Define the process used by lex tool for generating lexical analyzer.

Ans. Lex is a program designed to generate scanners, also known as tokenizers, which recognize lexical patterns in text. Lex is an acronym that stands for "lexical analyzer generator." Lex can perform simple transformations by itself but its main purpose is to facilitate lexical analysis, the processing of character sequences such as source code to produce symbol sequences called tokens for use as input to other programs such as parsers. **(2)**



- **lex.l** is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms lex.l to a C program known as lex.yy.c.
- **lex.yy.c** is compiled by the C compiler to a file called a.out.
- The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.
- **yylval** is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token. The attribute value can be numeric code, pointer to symbol table or nothing.

Q. 2. Consider the grammar given below ε =NULL

X → 0Y1 | 1Z2

Y → 2 | ε

Z → 1Y2 | 1 | ε

(a) Calculate first and follow for the non-terminals. **[5]**

(b) Create a predictive parsing table for the above grammar.

Also Show that this grammar is not LL(1). **[5]**

Ans.

$$X \rightarrow 0Y1 \mid 1Z2$$

$$Y \rightarrow 2/\epsilon$$

$$Z \rightarrow \mid Y2 \mid 1 \mid \epsilon$$

(a) Calculate first and follow

First (x) = {0,1} Follow (x) = {$}

First (y) = {2, ε} Follow (y) = {1,2}

First (z) = {1, ε} Follow (z) = {2}

Predictive Parsing table

	0	1	2	ε	$
X	X → 0Y1	X → 1Z2			
Y		Y → ε	Y → 2 Y → ε		Y → ε
Z		Z → 1Y2	Z → ε		

More than two entries exist
There is intersection. That is why
this grammar is not LL (1)

Q. 3. Consider the grammar

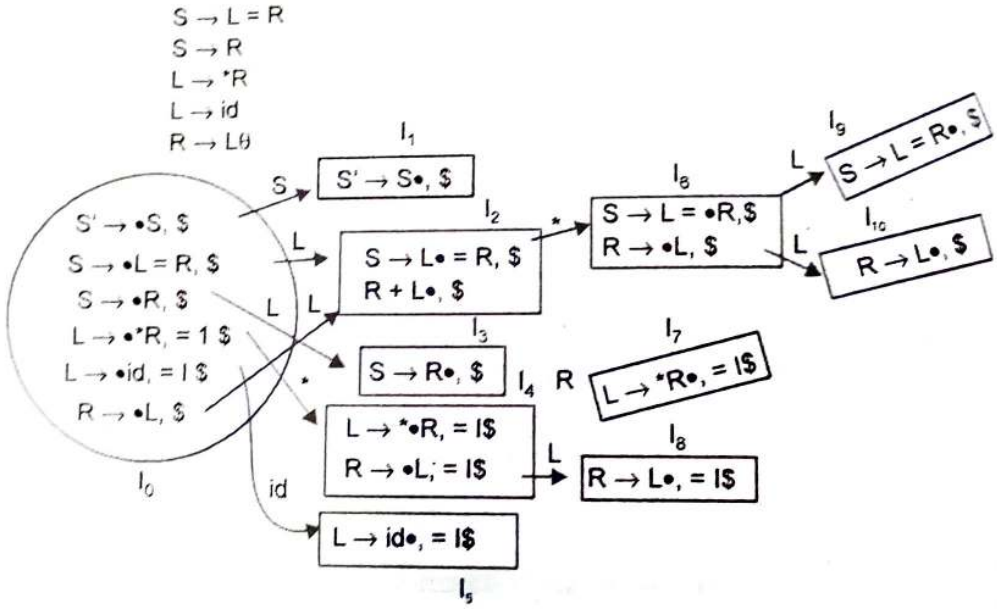
S → L = R | R

L → *R | id

R → L

Design LALR parser for this grammar. [10]

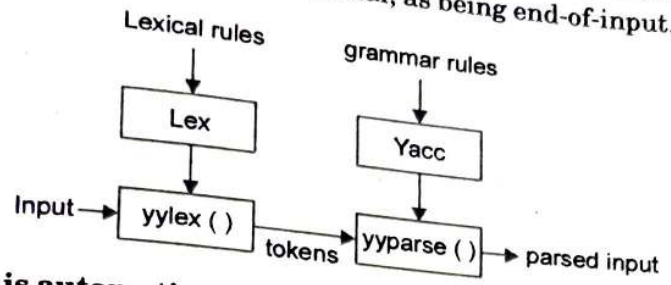
Ans. Design LALR Parser for this grammer.



Q. 4. (a) Describe the functioning of YACC with the help of an example. **(5)**

Ans. *Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and low-level routine to do the basic input. Yacc then generates a function to control the put process. This function, called a parser, calls the user-supplied low-level input outine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values*

of other actions. The Yacc utility transforms a specification of a context-free grammar into a C language function that implements an LR(1) parsing algorithm. Ambiguities in the grammar may be resolved using precedence rules within the specification. The C language function is named yyparse(), and calls yylex() to get input tokens, which are integer values. The value 0 is considered special, as being end-of-input.



The grammar is automatically checked by yacc to assure that:

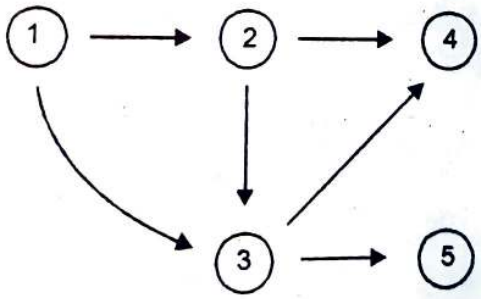
1. There are rules (productions) for all non-terminals.
2. All non-terminals must be reachable from the axiom.
3. The grammar is not ambiguous
4. The grammar is LALR(1) parsable.

The yacc program alerts us if any of these conditions do not hold. The first two are trivial to arrange.

Q. 4. (b) Define following terms:

(i) Topological Sort (ii) Dependency Graph (iii) Attribute (iv) Circular SDT (v) Evaluation order **[5]**

Ans. (i) Topological sort: Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG. For example: consider the graph given below:



A topological sorting of this graph is: 1 2 3 4 5

There are multiple topological sorting possible for a graph. For the graph given above one another topological sorting is: 1 2 3 5 4

(ii) Dependency graph: A dependency graph is a directed graph representing dependencies of several objects towards each other.

A program is a collection of statements, the ordering and scheduling of which depends on dependence constraints. Dependencies are broadly classified into two categories:

1. **Data Dependencies** : when statements compute data that are used by other statements

2. **Control Dependencies** : are those which arise from the ordered flow of control in a program

A dependence graph can be constructed by drawing edges connect dependent operations. These arcs impose a partial ordering among operations that prohibit a fully concurrent execution of a program

(iii) **Attribute**: It is the Information associated with a grammar symbol. These can be Computed using semantic rules associated with grammar rules. Types of Attributes:

- Synthesized attributes
- Inherited attributes

(iv) **Circular SDT**: Syntax-directed translation (SDT) refers to a method of compiler implementation where the source language translation is completely driven by the parser, i.e., based on the syntax of the language The parsing process and parse trees are used to direct semantic analysis and the translation of the source program.

(v) **Evaluation order:** It is the order in which the attributes are evaluated. If attribute X is calculated from attribute Y and Z, then the semantic rules for evaluating attributes Y and Z need to be performed BEFORE the semantic rule for calculating X.

END TERM EXAMINATION [MAY-JUNE. 2018]
SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS–302]

Time : 3 hrs. M.M. : 75

Note: Attempt any five questions including Q no. 1 which is compulsory.

Q. 1. Define the following terms and explain with example :-

(a) Lexeme, Token and Pattern.

Ans. Lexeme: A lexeme is a sequence of alphanumeric characters in a token. (3)
Lexemes are part of the input stream from which tokens are identified. An invalid or illegal token produces an error. A lexeme is one of the building blocks of language. It is is identified by the lexical analyzer as an instance of that token. These are Sequence of characters matched by PATTERN forming the TOKEN

Token: A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.

Pattern : The set of rule that define a TOKEN. A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.

Q. 1. (b) Activation Record. (3)

Ans. Activation record: An activation record is another name for Stack Frame. It's the data structure that composes a call stack.A general activation record consist of the following things:

- **Local variables:** hold the data that is local to the execution of the procedure.
- **Temporary values**: stores the values that arise in the evaluation of an expression.
- **Machine status:** holds the information about status of machine just before the function call.
- **Access link (optional):** refers to non-local data held in other activation records.
- **Control link (optional):** points to activation record of caller.
- **Return value:** used by the called procedure to return a value to calling procedure
- Actual parameters

(3)

Q. 1. (c) Bootstrapping.

Ans. Bootstapping: , **bootstrapping** is the technique for producing a self-compiling compiler that is, compiler written in the source programming language that it intends to compile. Bootstrapping a compiler has the following advantages:

- It is a non-trivial test of the language being compiled
- Compiler developers and bug reporting part of the community only need to know the language being compiled.
- Compiler development can be done in the higher-level language being compiled.
- Improvements to the compiler's back-end improve not only general-purpose programs but also the compiler itself.
- It is a comprehensive consistency check as it should be able to reproduce its own object code.

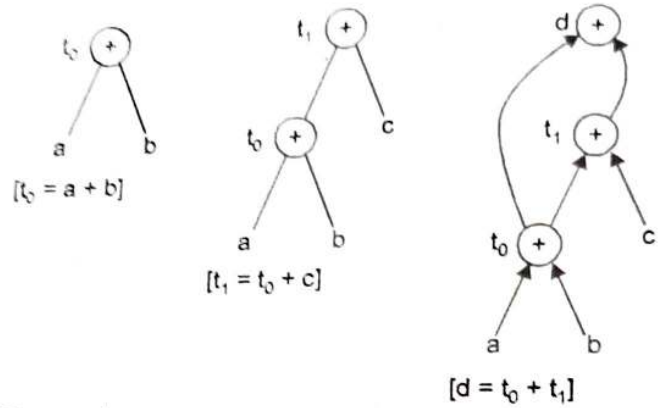
Q. 1. (d) Directed Acyclic Graph. (3)

Ans. DIRECTED ACYCLIC GRAPH: Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG can be understood here:

- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

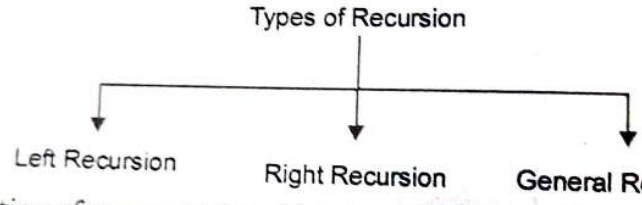
example:

$$t_0 = a + b$$
$$t_1 = t_0 + c$$
$$d = t_0 + t_1$$



Q. 1. (e) Left Recursion . (3)

Ans. Left recursion:Types of recursion



- A production of grammar is said to have **left recursion if the leftmost variable** of its RHS is same as variable of its LHS.A grammar **containing a production having left** recursion is called as a **Left Recursive Grammar. For example:**

$$S \rightarrow Sa / \epsilon$$

First eliminate **Left Recursion-**

If we have the left-recursive pair of productions-

$$A \rightarrow A\alpha / \beta$$

(Left Recursive Grammar)

where β does not begin with an A.

Then,

We can eliminate left recursion by replacing the pair of productions with-

$$A \rightarrow \beta A'$$
$$A \rightarrow \alpha A' / \epsilon$$

(Right Recursive Grammar)

• Thus, left recursion is eliminated by converting the grammar into a right recursive grammar having right recursion. This right recursive grammar functions same as left recursive grammar.

Q. 1. (f) Shift/Reduce and Reduce/Reduce Conflicts.

Ans. Shift/Reduce conflict: The Shift-Reduce Conflict is the most common type of conflict found in grammars. It is caused when the grammar allows a rule to be reduced for particular token, but, at the same time, allowing another rule to be shifted for that same token. As a result, the grammar is ambiguous since a program can be interpreted more than one way. This error is often caused by recursive grammar definitions where the system cannot determine when one rule is complete and another is just started. (3)

Reduce/Reduce Conflict: A Reduce-Reduce error is a caused when a grammar allows two or more different rules to be reduced at the same time, for the same token. When this happens, the grammar becomes ambiguous since a program can be interpreted more than one way. This error can be caused when the same rule is reached by more than one path.

Q. 1. (g) Back patching.

Ans. Backpatching: The syntax directed definition can be implemented in two or more passes when we have both synthesized attributes and inherited attributes: (3)

Build the tree first and then Walk the tree in the depth-first order.

The main difficulty with code generation in one pass is that we may not know the target of a branch when we generate code for flow of control statements.

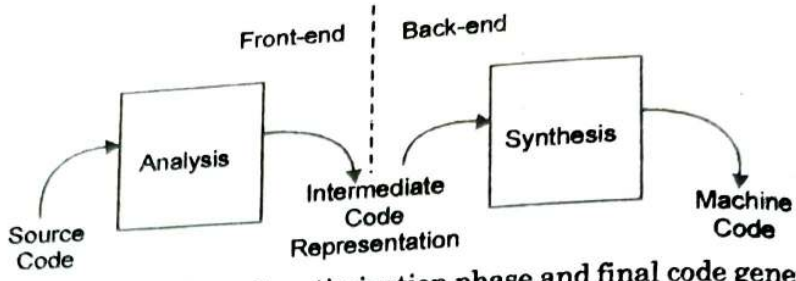
Backpatching is the technique to get around this problem. Generate branch instructions with empty targets When the target is known, fill in the label of the branch instructions. Backpatching is the process of leaving blank entries for the goto instruction where the target address is unkonown in the forward transfer in the first pass and filling these unknown in the second pass.

Q. 1. (h) Handle pruning. **(4)**

Ans. HANDLE PRUNING is the general approach used in shift-and-reduce parsing. A Handle is a substring that matches the body of a production. Handle reduction is a step in the reverse of rightmost derivation. A rightmost derivation in reverse can be obtained by handle pruning. The implementation of handle pruning involves the following data-structures:- a stack - to hold the grammar symbols; an input buffer that contains the remaining input and a table to decide handles.

Q. 2. (a) What do you mean by front end and back end of a compiler. **(4)**

Ans. The phases of a compiler are collected into front end and back end. The front end includes all analysis phases end the intermediate code generator that is why it is also known as Analysis phase. The front end analyzes the source program and produces intermediate code. The **analysis** phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.



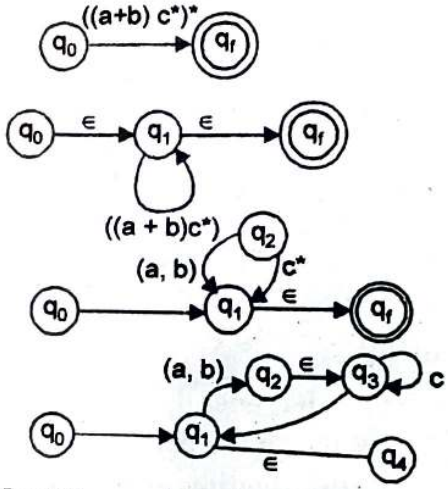
The back end includes the code optimization phase and final code generation phase. The back end synthesizes the target program from the intermediate code.This back end phase is also known as synthesis phase. The **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table. A compiler can have many phases and passes.

- **Pass** : A pass refers to the traversal of a compiler through the entire program.
- **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

Q. 2. (b) Construct a nondeterministic finite automaton (NFA) and then deterministic finite automaton (DFA) for the regular expression ((a/b)c*)*.(5)

Ans. $((a/b)\,c^*)^*$

$\Rightarrow$ $((a+b)\,c^*)^*$



Q. 2. (c) Explain Chomsky hierarchy of grammars. **(3.5)**

Ans. According to chomsky hierarchy, grammars are divided of 4 types:

Type 0 known as unrestricted grammar: Type-0 grammars include all formal grammars. Type 0 grammar language are recognized by turing machine. These languages are also known as the recursively enumerable languages.

Type 1 known as context sensitive grammar: Type-1 grammars generate the context-sensitive languages. The language generated by the grammar are recognized by the Linear Bound Automata.In Type 1

1. First of all Type 1 grammar should be Type 0.
2. Grammar Production in the form of

$$\alpha \rightarrow \beta$$

ount of symbol in α is less than or equal to β

Type 2 known as context free grammar: Type-2 grammars generate the context-free languages. The language generated by the grammar is recognized by a Non Deterministic Push down Automata. Type-2 grammars generate the context-free languages.

In Type 2,

1. First of all it should be Type 1.
2. Left hand side of production can have only one variable.

$$|\alpha| = 1.$$

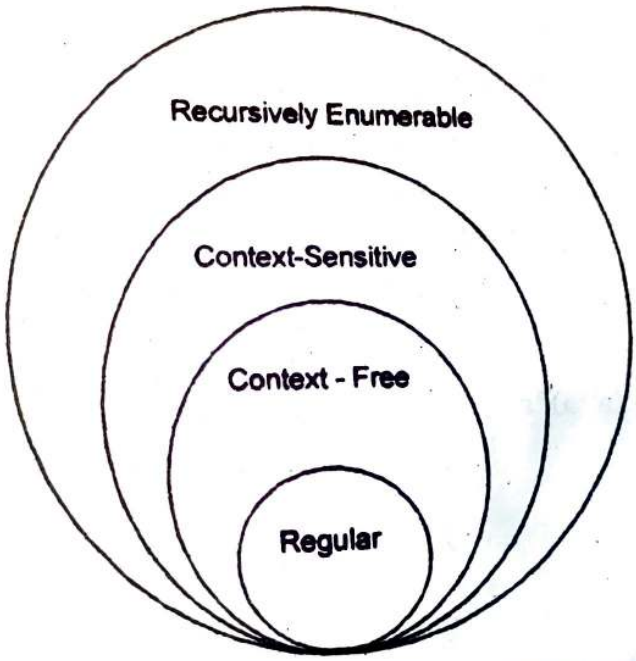
There is no restriction on β.

Type 3 Regular Grammar: Type-3 grammars generate the regular languages. These languages are exactly all languages that can be decided by a finite state automaton. Type 3 is most restricted form of grammar. Type 3 should be in the given form only :

$$V \rightarrow VT^*/T^*$$

(or)

$$V \rightarrow T^*V/T^*$$



Q. 3. (a) Show that the given Grammar is LL(1) but not SLR(1). **(6.5)**

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

Ans. To show that it is not SLR(1) but LL(1), consider the LR(0) items in the initial set for the augmented grammar:

$$S' \rightarrow S$$
$$S \rightarrow AaAb \mid BbBa$$
$$A \rightarrow \varepsilon$$
$$B \rightarrow \varepsilon$$
$$I_0 : S' \rightarrow \bullet S$$
$$S \rightarrow \bullet AaAb$$
$$S \rightarrow \bullet BbBa$$

$A \rightarrow \bullet$

$B \rightarrow \bullet$

And consider that FOLLOW(A) = FOLLOW(B) = {a,b}.

In the first state we have a reduce/reduce conflict, since the FOLLOW of A and B are same on the input terminal 'a' of the token we don't know whether to retrace the rule $A \rightarrow \varepsilon$ or $B \rightarrow \varepsilon$. Hence in the LR parsing table we have two conflicting entries for action[0,a]. Same holds for action[0,b].

Now consider the LL(1) parser for the above grammar.

FIRST(S) = {a,b}

FIRST (A) = FIRST (B) = {ε}

FIRST(A) = FIRST(B)= {a, b}

FOLLOW(S) = {$}

The parse table is

	a	b	$
S	S → AaAb	S → BbBa	
A	A → ε	A → ε	
B	B → ε	B → ε.	

No conflicts, so the grammar is LL(1).

Q. 3. (b) Give parse tree's and derivation's (leftmost and rightmost) for the grammar and input string given below. . (6)

E → E+T / T

T → T*F / F

F → (E) / a

Input string: a + (a*a)*a

Ans.

$E \Rightarrow E + T/T$

$T \rightarrow T * F/F$

$F \rightarrow (E) \mid a$

Input strong: a + (a * a) + a

Left Most derivation

$E \rightarrow E + T$

$\rightarrow T + T$

$\rightarrow F + T$

$\rightarrow a + T$

$\rightarrow a + T*F$

$\rightarrow a + F * F$

$\rightarrow a + (E) * F$

$\rightarrow a + (T) * F$

$\rightarrow a + (T* F) * F$

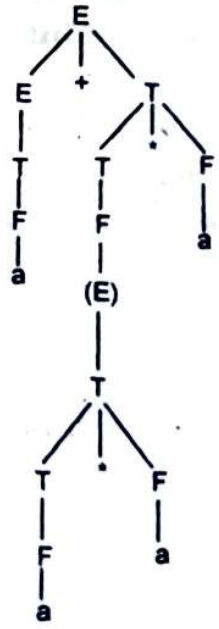
$\rightarrow a + (F * F) * F$

$\rightarrow a + (a * F) * F$

$\rightarrow a + (a * a) * F$

$\rightarrow a + (a * a) * a$

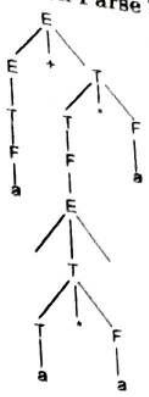
Left-most derivation Parse Table



Right most derivation

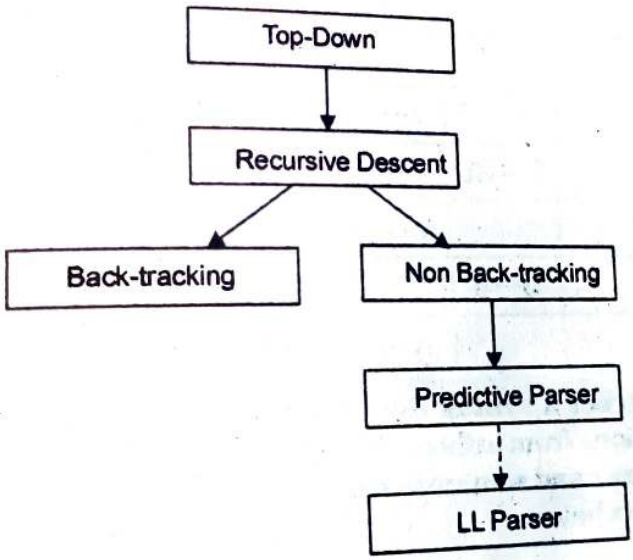
$E \rightarrow E + T$
$\rightarrow E + T*F$
$\rightarrow E + T*a$
$\rightarrow E + F*a$
$\rightarrow E + (E)*a$
$\rightarrow E + (T)*a$
$\rightarrow E + (T*F)*a$
$\rightarrow E + (T*a)*a$
$\rightarrow E + (a*a)*a$
$\rightarrow T + (a*a)*a$
$\rightarrow F + (a*a)*a$
$\rightarrow a + (a*a)*a$

Right most derivation Parse Tree



Q. 4. (a) Explain Recursive descent Parser with example. (5.5)

Ans. Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. It recursively parses the input string means on the basis of using one instance of a command or event to generate another.



Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them.

Q. 4. (b) Construct the canonical parsing table for the grammar given below. (7)

$S \rightarrow aA\,Be$

$A \rightarrow A\,bc \mid b$

$B \rightarrow d$

Ans.

S0: $\bullet\, S \rightarrow \bullet a\,A\,B\,e$

S1: $S \rightarrow a \bullet A\,B\,e$

$A \rightarrow \bullet A b c \mid \bullet b$

$A \rightarrow \bullet A b c \mid \bullet b$

S2: $A \rightarrow b \bullet$

S3: $S \rightarrow aA \bullet B e$

$B \rightarrow \bullet d$

$A \rightarrow A \bullet b c$

S4: $A \rightarrow Ab \bullet c$

S5: $A \rightarrow Abc \bullet$

S6: $S \rightarrow aA \bullet Be$

$B \rightarrow \bullet d$

S7: $B \rightarrow d \bullet$

S8: $S \rightarrow aAB \bullet e$

S9: $S \rightarrow aABe \bullet$

S10: $S' \rightarrow S \bullet$

	(	)	a	b	$	S	A	B
0	1		9				4	
1	1		9			2	4	
2		3						
3		S→(S)			S→(S)			
4			7	8				5
5		S→AB		6	S→AB			
6		B→Bb		B→Bb	B→Bb			
7			A→Aa	A→Aa				
8		B→b		B→b	B→b			
9			A→a	A→a				

Q. 5. (a) Construct a syntax directed translation scheme that translates arithmetic expressions from infix to postfix notation. The solution should include context free grammar and semantic rules. Show the application of your scheme with the input given below. **(6.5)**

Ans. 1+2/3-4*5

1. step 1: Parenthesize (using standard precedence) to get (1+(2/3))-(4*5)

2. Apply the above rules to calculate P{(1+(2/3))-(4*5)}, where P{X} means "convert the infix expression X to postfix".

A. P {(1 + (2/3))-(4*5)]

B. P{1 + (2/3))} P {(4*5)}-

C. P{1 + (2/3)} P {4*5}-

D. P {1} P {2/3} + P{4} P {5} *-

E. 1P{2} P {3}/ + 4 5*–

F. 1 2 3 / + 4 5 *-

Q. 5. (b) **Translate the following expression to quadruple and triple representation. A = - B * (C+D)/E** **(6)**

Ans. Quadruple representation

t_1 = Uminus B
t_2 = C
$t_3 = t_2 + D$
$t_4 = t_1 * t_3$
t_5 = e
$A = t_4/t_5$

	OP	Arg 1	Arg 2	result
(0)	Uminus	B		t_1
(1)		C		t_2
(2)	+	t_2	D	t_3
(3)	*	t_1	t_3	t_4
(4)		e		t_5
(5)	/	t_4	t_5	A

Triple Representation

Number	OP	Arg 1	Arg 2
(0)	Uminus	B	
(1)		C	
(2)	+	(1)	D
(3)	*	(0)	(2)
(4)		e	
(5)	/	(3)	(4)

Q. 6. (a) **What are the uses of symbol table in different phases of compiler design?** **(3)**

Ans. Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variable i.e. it stores information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

It is used by various phases of compiler as follows :-

1. **Lexical Analysis:** Creates new table entries in the table, example like entries about token.

2. **Syntax Analysis:** Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.

3. **Semantic Analysis:** Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.

4. **Intermediate Code generation:** Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.

5. Code Optimization: Uses information present in symbol table for machine dependent optimization.

6. Target Code generation: Generates code by using address information o identifier present in the table.

Each entry in symbol table is associated with attributes that support compiler i different phases.

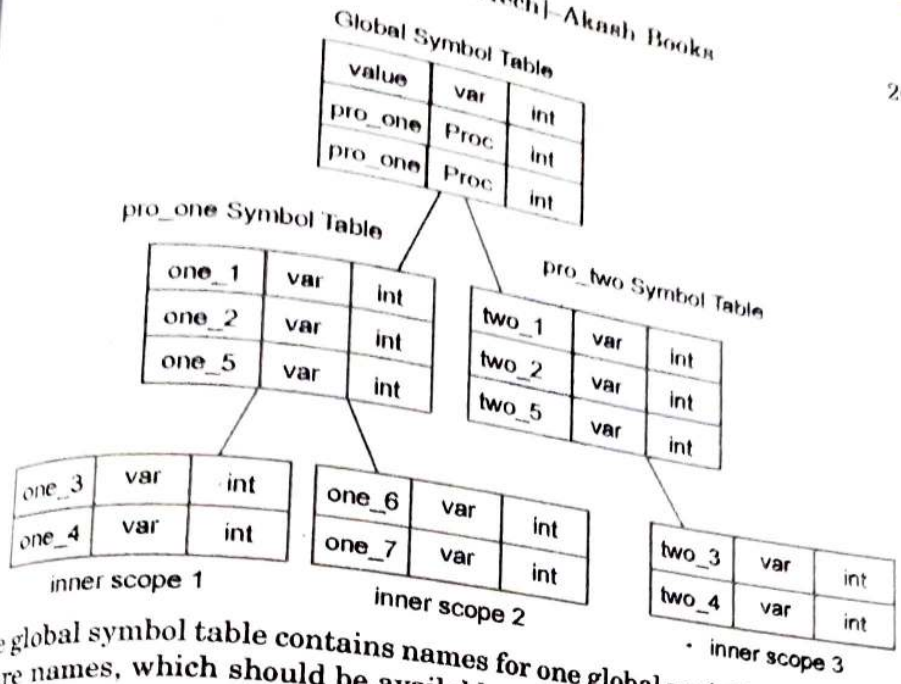
Q. 6. (b) Explain how scope information is represented by symbol table. (4

Ans. A compiler maintains two types of symbol tables: a **global symbol table** whic can be accessed by all the procedures and **scope symbol tables** that are created fo each scope in the program. To determine the scope of a name, symbol tables are arrange in hierarchical structure as shown in the example below:

```
int value=10;
void pro_one()
{
int one_1;
int one_2;
{                \
int one_3;       |_ inner scope 1
int one_4;       |
}                /
int one_5;
{
                 \
int one_6;       |_ inner scope 2
int one_7;       |
}                /
}

void pro_two()
{
int two_1;
int two_2;
{                \
                 |_ inner scope 3
                 |
                 /
5;
```

The above program can be represented in a hierarchical structure of symbol tables:



The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available in pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- first a symbol will be searched in the current scope, i.e. current symbol table.
- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- either the name is found or global symbol table has been searched for the name.

Q. 6. (c) Explain different data structures for symbol table implementation and compare them. **(5.5)**

Ans. Following are commonly used data structure for implementing symbol table :-

1. **List –**
   - In this method, an array is used to store names and associated information.
   - A pointer **"available"** is maintained at end of all stored records and new names are added in the order as they arrive
   - To search for a name we start from beginning of list till available pointer and if not found we get an error **"use of undeclared name"**
   - While inserting a new name we must ensure that it is not already present otherwise error occurs i.e. **"Multiple defined name"**
   - Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
   - Advantage is that it takes minimum amount of space.
2. **Linked List –**
   - This implementation is using linked list. A link field is added to each record.
   - Searching of names is done in order pointed by link of link field.
   - A pointer **"First"** is maintained to point to first record of symbol table.
   - Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

3. Hash Table –

- In hashing scheme two tables are maintained – a hash table and symbol tabl and is the most commonly used method to implement symbol tables..
- A hash table is an array with index range: 0 to tablesize – 1.These entries ar pointer pointing to names of symbol table.
- To search for a name we use hash function that will result in any integer betwee 0 to tablesize – 1.
- Insertion and lookup can be made very fast – O(1).
- Advantage is quick search is possible and disadvantage is that hashing complicated to implement.

4. Binary Search Tree –

- Another approach to complement symbol table is to use binary search tree i. we add two link fields i.e. left and right child.
- All names are created as child of root node that always follow the property binary search tree.
- Insertion and lookup are $O(\log_2 n)$ on average.

Q. 7. (a) Why we need code optimization? Explain Strength reduction, Su expression elimination, dead code elimination and loop optimization technique (6.5)

Ans. Code optimization: The code optimization in the synthesis phase is a progra transformation technique, which tries to improve the intermediate code by making consume fewer resources so that faster-running machine code will result. Compil optimizing process should meet the following objectives :

- The optimization must be correct, it must not, in any way, change the meanir of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

Optimization of the code is often performed at the end of the development sta since it reduces readability and adds code that is used to increase the performance.

Strength reduction: In compiler construction, **strength reduction is a compil** optimization where expensive operations are replaced with equivalent but less expensi operations. The classic example of strength reduction converts "strong" multiplicatio inside a loop into "weaker" additions – something that frequently occurs in arr addressing.

Examples of strength reduction include:

- replacing a multiplication within a loop with an addition
- replacing an exponentiation within a loop with a multiplication

```
c = 7;
for (i = 0; i < N; i++)
{
    y[i] = c* i;
}
```

can be replaced with successive weaker additions

```
c = 7;
```

```
k = 0;
for (i = 0; i < n; i ++)

  y[i] = k ;
  k = k + c ;
}
```

Sub expression Elimination:

Subexpression elimination is a compiler optimization that searches for instances identical expressions (i.e., they all evaluate to the same value), and analyzes whether worthwhile replacing them with a single variable holding the computed value. The sibility to perform sub expression elimination is based on available pression analysis. Compiler writers distinguish two kinds of CSE:

- **local common subexpression elimination** works within a single basic block
- **global common subexpression elimination** works on an entire procedure,

Both kinds rely on data flow analysis of which expressions are available at which nts in a program.

For example:

```
a = b * c + g;
d = b * c * e;
```

This code will be transformed to:

```
tmp = b * c;
a = tmp + g;
d = tmp * e;
```

if the cost of storing and retrieving tmp is less than the cost of calculating b*c an ra time.

Dead code Elimination: Dead code elimination is a compiler optimization to move code which does not affect the program results. It is also known as dead code moval, dead code stripping, or dead code strip. Dead code elimination shrinks program ze, an important consideration in some contexts, and it allows the running program to void executing irrelevant operations, which reduces its running time. It can also enable rther optimizations by simplifying program structure. *Dead code* includes code that n never be executed and code that only affects *dead variables* that is, irrelevant to the ogram. For example:

```
int global;
void f ()
{
  int i;
  i = 1;              /* dead store */
  global = 1;         /* dead store */
  global = 2;
  return;
  global = 3;         /* unreachable */
}
```

In the above example. the value assigned to i is never use.

The code fragment after dead code elimination:

```
int global ;
void f ()
{
global = 2;
return ;
}
```

Loop optimization: Loop optimization is the process of increasing execut speed and reducing the overheads associated with loops. It plays an important role improving cache performance and making effective use of para processing capabilities. It becomes necessary to optimize the loops in order to save C cycles and memory. Loops can be optimized by the following techniques:

- **Invariant code** : A fragment of code that resides in the loop and computes same value at each iteration is called a loop-invariant code. This code can be moved of the loop by saving it to be computed only once, rather than with each iteration.
- **Induction analysis** : A variable is called an induction variable if its value altered within the loop by a loop-invariant value.
- **Strength reduction** : There are expressions that consume more CPU cycles, tim and memory. These expressions should be replaced with cheaper expressions witho compromising the output of expression.

Q. 7. (b) Write short notes on:

(i) Problems in code generation

Ans. Problems in code generation: Code generator converts the intermedia representation of source code into a form that can be readily executed by the machine code generator is expected to generate a correct code.

Issues in code generation:

Input to code generator:The input to code generator is the intermediate co generated by the front end, along with information in the symbol table that determin the run-time addresses of the data-objects denoted by the names in the intermedia representation assuming that the they are free from all of syntactic and state semant errors, the necessary type checking has taken place and the type-conversion operato have been inserted wherever necessary.

Target program –

Target program is the output of the code generator. The output may be absolut machine language, relocatable machine language, assembly language.

- Absolute machine language as an output has advantages that it can be placed i a fixed memory location and can be immediately executed.
- Relocatable machine language as an output allows subprograms and subroutine to be compiled separately. Relocatable object modules can be linked together and loade by linking loader.

• Assembly language as an output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code.

Memory Management –

Mapping the names in the source program to addresses of data objects is done by the front end and the code generator. A name in the three address statement refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

Instruction selection –

Selecting best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

Register allocation issues –

Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:

1. During **Register allocation –** we select only those set of variables that will reside in the registers at each point in the program.

2. During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

Evaluation order –

The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in general case is a difficult NP-complete problem.

Approaches to code generation issues: Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:

- Correct
- Easily maintainable
- Testable
- Maintainable

Q. 7. (b)(ii) Uses of Basic Blocks. **(3)**

Ans. Basic block is a set of statements which always executes in a sequence one after the other without getting halt in the middle or any possibility of branching. Basic blocks do not contain any kind of jump statements in them. All the statements in a basic block gets execute in the same order as they appear in the block without losing the flow control of the program. A **basic block is a straight-line code sequence with no** branches in except to the entry and no branches out except at the exit. The code in a basic block has:

SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS-302]

M.M. : 30

Time : 1.5 hrs.

Note: *Attempt Q.No. 1 which is compulsory and any two other questions from remaining*

Q.1. (a) How dependency graph effects the evaluation order in syntax directed translation? (2)

Ans. Order of evaluating attributes is important. Dependency graph is general rule for ordering. Dependency graph is a directed graph and it shows interdependencies between the attributes. If an attribute 'b' at a node depends on an attribute 'c', then the semantic rule for 'b' at that node must be evaluated after the semantic rule that defines 'c'.

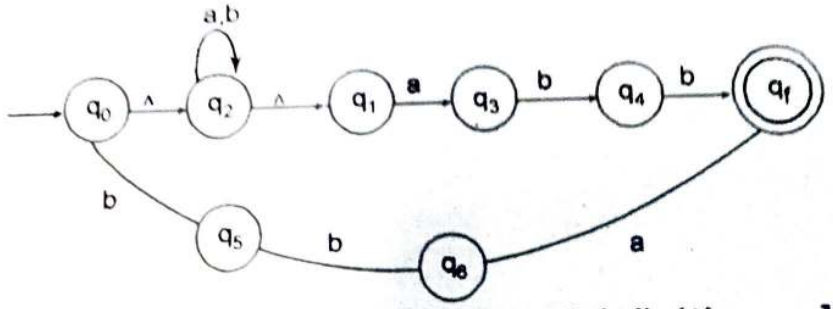
Q. 1. (b) "Lexical Analysis is the most time consuming phase in compilation" justify this statement. (2)

Ans. Lexical Analysis is the first phase when compiler scans the source code. This process can be left to right, character by character, and group these characters into tokens. Here, the character stream from the source program is grouped in meaningful sequences by identifying the tokens. It makes the entry of the corresponding tickets into the symbol table and passes that token to next phase. The primary functions of this phase are:

- Identify the lexical units in a source code
- Classify lexical units into classes like constants, reserved words, and enter them in different tables. It will Ignore comments in the source program
- Identify token which is not a part of the language

Q. 1. (c) Create NDFA for the regular expression (a + b)* abb + bba. (2)

Ans. NDFA for regular expression : (a + b)* abb + bba



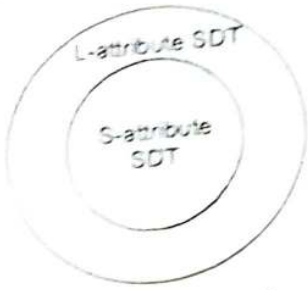
Q. 1. (d) Differentiate between L-Attributed definition and S-Attributed definition. (2)

Ans. S-Attributed SDT: If an SDT **uses only synthesized attributes, it is called as** *S-attributed SDT.*

- *S-attributed SDTs* are evaluated **in bottom-up parsing, as the values of the parent** nodes depend upon the values of the **child nodes.**
- Semantic actions are placed in **rightmost place of RHS.**
- ***L-attributed SDT:*** If an SDT **uses both synthesized attributes and inherited** attributes with a restriction that i**nherited attribute can inherit values from left siblings** only, it is called as L-attributed **SDT.**
- Attributes in L-attributed **SDTs are evaluated by depth-first and left-to-right** parsing manner.

- Semantic actions are placed anywhere in RHS.

If a definition is S-attributed, then it is also L-attributed but **NOT** vice-versa.



Q. 1. (e) Remove left recursion from the following grammer: **(2)**

$S \rightarrow AaAb \mid b$

$A \rightarrow Ab \mid a \mid ab$

Ans. There is no left recursion in $S \rightarrow AaAb/b$. Left recursion is only in:

$A \rightarrow Aa \mid a \mid ab$

$A \rightarrow aA' \mid abA'$

$A' \rightarrow bA' / \epsilon$

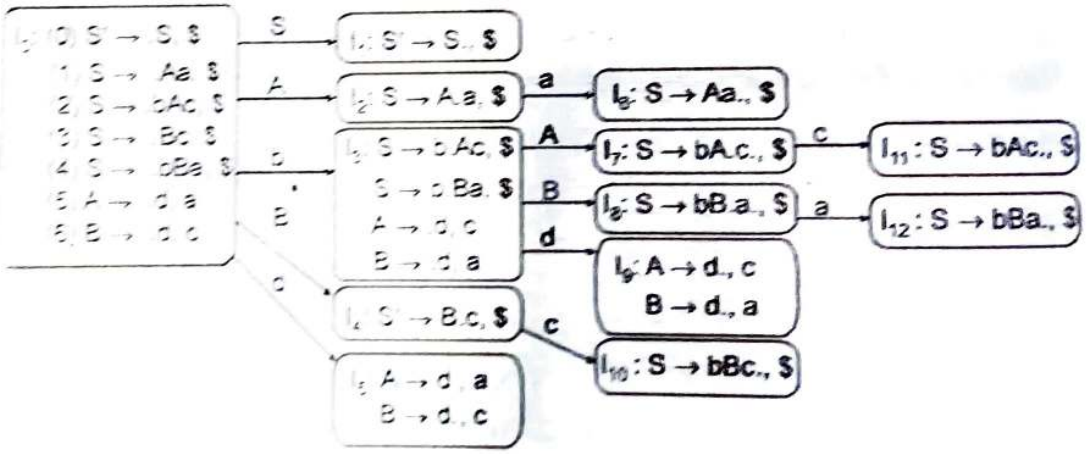
Q. 2. Prove that following grammar is CLR (1) but not LALR(1) **(10)**

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$

Ans.



Based on the state diagram, we **derive the L.R (1) parsing table as follows:**

State	Action					Goto		
	a	b	c	d	$	S	A	B
0		s_3		s_4		1	2	4
1					acc			
2	s_6							
3				s_9			7	8
4			s_{10}					

5	r_5	r_6			
6			s_{11}		
7	s10				
8	r_6		r_5		
9				r_3	
10				r_2	
11				r_4	
12					

Since there are no muliple actions in any entry, the given grammar is LR(1). However, when obtaining the LALR(1) parsing table by merging states, we will merge states l_5 and l_9, and the resulting state will be as follows:

(2) l_{5+9} A → d., a/c

B → d, a/c

It is basically a reducc-reduce conflict. So, the given grammar is not LALR(1).

Q. 3. (a) Calculate first and follow for all non-terminals for the following grammar. **(6)**

E→E+T | T

T→TF | F

(10) **F→F* | a | b**

Ans. First remove left recursion from the above grammar.

∴

E	→	TE′
E′	→	+TE′/∈
T	→	FT′
T′	→	*FT′/∈
F	→	aF′/bF′
F′	→	*F′/∈

Now calculate First and Follow.

First (E)	=	First (T) = First (F) = {a, b}
First (E′)	=	{+, ∈}
First (T′)	=	{*, ∈}
First (F′)	=	{*, ∈}
Follow(E)	=	{$}
Follow (E′)	=	Follow (E) = {$}
Follow (T)	=	First (E′) ∪ Follow (E)
	=	{+, $}
Follow (T′)	=	Follow (T)
	=	{+, $}
Follow (F)	=	First (T′) ∪ Follow(T)
	=	{*, +, $}
Follow (F′)	=	Follow (F)
	=	{*, +, $}

Q. 3. (b) Create a predictive parsing table for the grammar shown in part a. **(3)**

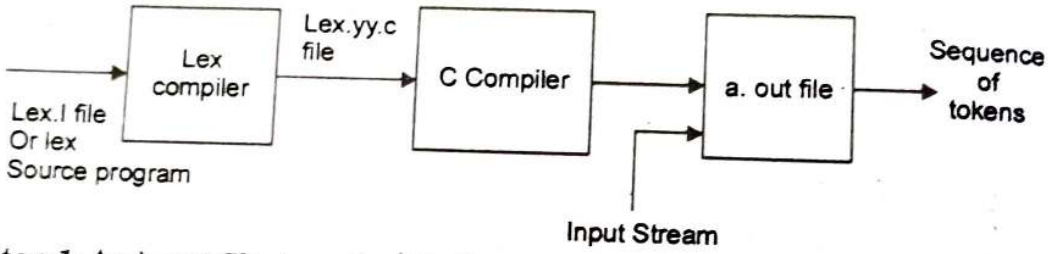
Ans. Predictive Parsing table for previous question

	+	*	a	b	$
E			$E \to TE'$	$E \to TE'$	
E'	$E' \to TE'$				$E' \to \epsilon$
T			$T \to FT'$	$T \to FT'$	
T'	$T' \to \epsilon$	$T' \to *FT'$			$T' \to \epsilon$
F			$F \to aF'$	$F \to bF'$	
F'	$F' \to \epsilon$	$F' \to *F'$ $F' \to \epsilon$			$F' \to \epsilon$

↑ Mutliple enteries (more than one entry)

Q. 4. (a) How lex (flex)tool can be used to create lexical analyzer? Explain with the help of an example. **(5)**

Ans. FLEX (fast lexical analyzer generator) is a tool/computer program for generating lexical analyzers.



Step 1: An input file describes the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.

Step 2: The C complier compile lex.yy.c file into an executable file called a.out.

Step 3: The output file a.out take a stream of input characters and produce a stream of tokens.

Program Structure: In the input file, there are 3 sections:

1. Definition Section: The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in "%{ %}" brackets. Anything written in this brackets is copied directly to the file lex.yy.c

```
%{
    //Definitions
%}
```

2. **Rules Section:** The rules section contains a series of rules in the form: *pattern action* and pattern must be unintended and action begin on the same line in { } brackets. The rule section is enclosed in "%% %%".

```
%%
    pattern action
%%
```

3. User Code Section: This section contain C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

How to run the program: To run the program, it should be first saved with the extension .l or .lex. Run the below commands on terminal in order to run the program file.

END TERM EXAMINATION [MAY. 2019]
SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS–302]

Time : 3 hrs. **M.M. : 75**

Note :- *Attempt five questions in all including Question No. 1 which is compulsory. Select one question from each unit.*

Q.1. Attempt any five parts from the following:

Q.1. (a) What do you mean by 'pass' in a compiler? Differentiate between a multipass and single pass compiler. **(5)**

Ans. A Compiler pass refers to the traversal of a compiler through the entire program.

Multi-Pass Compiler	One Pass Compiler
1. It reads the program Several times, each time transforming it into different forms.	1. It reads program only one and translate it at same time.
2. They are "slower", As, more number of passes means more execution time	2. They are faster
3. Better code optimization and code generations	3. Less efficient code optimization and code generation.
4. It is also called wide compiler. As they can scan each and every portion of the program.	4. It is also called narrow compiles as it has limited scope.
5. Memory occupied by one pass can be reused by subsequent pass therefore, small memory is required by compiler	5. Large memory is required by compiler
6. E.g. Modula-2 Lang uses Multi-pass compilation	6. E.g. Pascal and Capital-lang uses one pass compilation.

Q. 1. (b) Why do we need to have lexical analyzer generators? What are its advantages? **(5)**

Ans. Lex is a program designed to generate scanners, also known as tokenizers, which recognize lexical: patterns in text. Lex is an *acronym* that stands for "lexical analyzer generator." It is intended primarily for Unix-based systems.Its main purpose is to facilitate lexical analysis, the processing of character sequences such as *source code* to produce symbol sequences called tokens for use as input to other programs such as *parsers*. Lex can be used with a parser generator to perform lexical analysis. It is easy, for example, to interface Lex and Yacc, an *open source* program that generates code for the parser in the C programming language.

Q. 1. (c) What are the responsibilities of Loader, Linker and Assembler in the compiler environment? **(5)**

Ans. Assembler: An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

Linker: Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/ routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

Loader: Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution. (5)

Q. 1. (d) Explain cross compiler with a suitable example.

Ans. Refer to Q.4. (a) First Term Examination 2017. (Page No. 2017).

Q. 1. (e) Write a regular expression for each of the following sets of binary strings. Use only the basic operations. (5)

(i) contains the substring 110

(ii) doesn't contain 110 as prefix

Ans. (i) $(1+0)^*110(1+0)^*$

(ii) $0^*1(1+0)^*+10^*(1+0)^*+(1+0)^*$

Q.1. (f) Left factor the following grammar: $\mathbf{E \rightarrow E + T \mid E}$, $\mathbf{T \rightarrow int \mid (E)}$ (5)

Ans. Left factor of above grammar :

$$E \rightarrow EE'$$
$$E' \rightarrow +T/\epsilon$$
$$T \rightarrow int(E)$$

Q.1. (g) Differentiate between SDD and SDT. (5)

Ans. SDT: It is a kind of notation in which each production of CFG is associated with a set of semantic rules or actions and each grammar symbol is associated with a set of attributes. Therefore the grammar and the semantic actions combine to make a Syntax Directed Translation.

It is of two types:

1. **Synthesized Translation:** These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

$S \rightarrow ABC$

If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

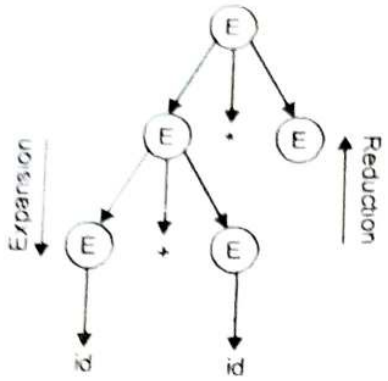
As in our previous example ($E \rightarrow E + T$), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

2. **Inherited Translation:** In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production.

$S \rightarrow ABC$

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Expansion: When a non-terminal is expanded to terminals as per a grammatical rule



Reduction : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks. Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis). Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

For example:

int value = 5;

<type, "integer">

<presentvalue, "5">

For every production, we attach a semantic rule.

SDD (Syntax Directed Defination): A CFG where each grammar production A –> α is associated with a set of semantic rules of the form b = f(c1, c2, ..., ck); where: b is a synthesized attribute of A or an inherited attribute of one of the grammar symbols in α c1, c2, ... are attributes of the symbols used in the production.

Synthesized and Inherited Attributes

- Attributes are values computed **at the nodes of a parse tree.**
- *Synthesized attributes* are values **that are computed at a node *N* in a parse tree** from attribute values of the **children of *N* and perhaps *N* itself. The identifiers** $$, $1, $2, etc., in Yacc actions **are synthesized attributes. Synthesized** attributes can be easily computed **by a shift-reduce parser that keeps the values** of the attributes on the parsing **stack.**
- An SDD is *S-attributed* if every **attribute is** synthesized.
- *Inherited attributes* are values **that are computed at a node *N* in a parse tree** from attribute values of the **parent of *N*, the siblings of *N*, and *N* itself.**
- An SDD is *L-attributed* is every **attribute is either synthesized or inherited from** the left.

UNIT-I

Q. 2. (a) Prove that the following **grammar is LL(1) by constructing predictive** parsing tale. **(5)**

Prove that grammar is LL(1)

$N \rightarrow AB$	$N \rightarrow BA$
$A \rightarrow a$	$A \rightarrow CAC$
$B \rightarrow b$	$B \rightarrow CBC$
$C \rightarrow a$	

Ans.

A
B
C
N

Q. 2. (b

S → R

R → a

Ans. S

Calcul

State
0
1
2
3

Ans. Calculate First and Follow

First (N) = First (A) ∪ First (B) = {a, b}
First (A) = {a}
First (B) = {a, b}
First (C) = {a}

Follow (N) = {$}
Follow (A) = {a, b, $}
Follow (B) = {a, b, $}
Follow (C) = {a, b, $}

	a	b	$
A	A → a A → cAC		
B	B → CBC	B → b	
C	C → a		
N	N → AB	N → BA	

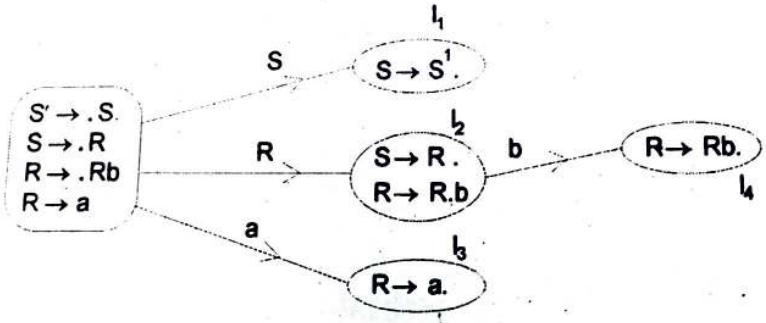
Multiple entries in A. So the grammar is not LL (1)

Q. 2. (b) Write SLR Parser for following grammar **(7.5)**

S → R R → Rb
R → a

Ans. SLR Parser for the following:

S → R, R → Rb/a



S → R
R → Rb/a

Calculate First and Follow

First (S) = First (R) = {a}
Follow (S) = { $ }
Follow (S) = { b } ∪ Follow (S)
= {b, $}

States	Action			go to	
	a	b	$	R	S
0	S_3	—	—	2	1
1			accept		
2		S_4			
3		r_3	r_3		
4		r_2	r_2		

Q.3. (a) What is operator precedence grammar? Construct the operator precedence parser for following. (7.5)

S→aIJh **I→IbSe | c** **(Incorrect Question)**

J→KLkr **K→d**

L→p

Consider the following grammar-

$$S \rightarrow (\ L\ ) \mid a$$

$$L \rightarrow L\ ,\ S \mid S$$

Ans. An **operator precedence grammar** is a kind of grammar for formal languages. An **operator precedence grammar** is a context-free grammar that has the property that no production has either an empty right-hand side (null productions) or two adjacent non-terminals in its right-hand side.

Operator precendence question:

Construct the operator precedence parser and parse the string (a , (a , a)).

	a	(	)	,	$
a		>	>	>	>
(	<	>	>	>	>
)	<	>	>	>	>
,	<	<	>	>	>
$	<	<	<	<	

Operator Precedence Table

For parsing the given string **(a , (a , a)),** insert **$** symbol at both ends of the string as-

$ (a , (a , a)) $

insert precedence operators between the string symbols as-

$$\$ < (< a > , < (< a > , < a >) >) > \$$$

Now scan and parse the string as-

$\$ < (\underline{< a >} , < (< a > , < a >) >) > \$$

$\$ < (S , < (\underline{< a >} , < a >) >) > \$$

$\$ < (S , < (S , \underline{< a >}) >) > \$$

$\$ < (S , \underline{< (S , S) >}) > \$$

$\$ < (S , \underline{< (L , S) >}) > \$$

$\$ < (S , \underline{< (L) >}) > \$$

$\$ \underline{< (S , S) >} \$$

$\$ \underline{< (L , S) >} \$$

$\underline{< (L) >} \$$

$\underline{< S >} \$$

$\$$

Q.3. (b) Is following grammar left recursive? If yes remove it. **(5)**

S → A B **A → C B | b**
C → S a **B → b**

Ans.

	S	→	AB
	A	→	CB/b
	C	→	Sa
	B	→	b

Here Indirect left recursion exists

S	→	CBB/bB
C	→	Sa
B	→	b

Substitute C → Sa in place of S

S	→	SaBB/bB
B	→	b

Now remove left recursion

S	→	aBBs′/bB
S′	→	∈
B	→	b

UNIT–II

Q. 4. (a) Consider the following SDTS (Syntax Directed Translation Scheme): **(7.5)**

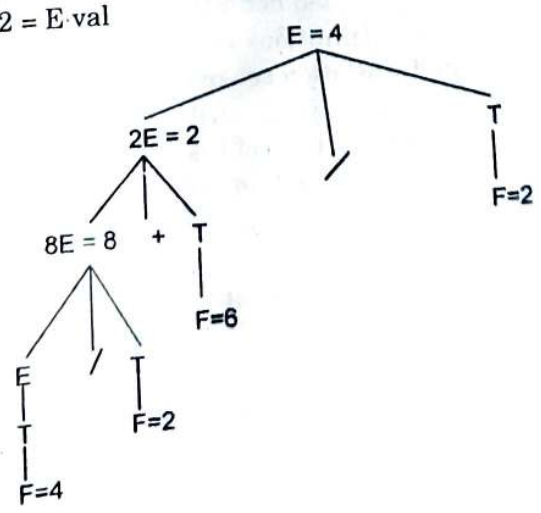
E→E/T **{E.val=E_1.val*T.val}**
E→E+T **{E.val=E_1.val-T.val}**
E→T **{E.val=T.val}**
T→F **{T.val=F.val}**
F→2/4/6 **{F.val=2} {F.val=4} {F.val=6}**

Using the above SDTS, construct the parse tree for the expression: 4/2+6/2 and evaluate 'E.val' at root of the tree.

Ans.

E	→	E/T	(E.val = E_1.val * T.val)
E	→	E + T	(E·val = E_1·val – T·val)
E	→	T	(E·val = T·val)
T	→	F	(T·val = F·val)
F	→	2/4/6	(F·val = 2) (F·val = 4) (F·val = 6)

uate : 4/2 + 6/2 = E·val



The E·val = 4

Indirect Triple Representation-

	Statement
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

Location	Op	Arg 1	Arg 2
(0)	1↑	e	f
(1)	*	b	e
(2)	/	(1)	(0)
(3)	*	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

Q. 5. (b) Write three address code for following code segment. **(7.5)**

```
int a[10], b[10], dot_prod, i;
int* a1; int*b1;
dot_prod=0;
a1=a;
b1-b;
for(i=0; i<10;i++)
        dot_prod+=*a 1 ++* *b 1++;
```

Ans. dot_prod = 0

```
                a1 = &a
                b1 = &b
                 i = 0
L1 : if (i ≥ 10) goto L2
                T3 = *a1
                T4 = a1 + 1
                a1 = T4
                T5 = *b1
                T6 = b1 + 1
                b1 = T6
                T7 = T3 * T5
                T8 = dot_prod + T7
          dot_prod = T8
                T9 = i + 1
                 i = T9
              goto L1
           L2 : Exit
```

UNIT-III

Q. 6. (a) What is symbol table? Why symbol table is required? What type of data structures are used for its implementation? **(7.5)**

Ans. Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

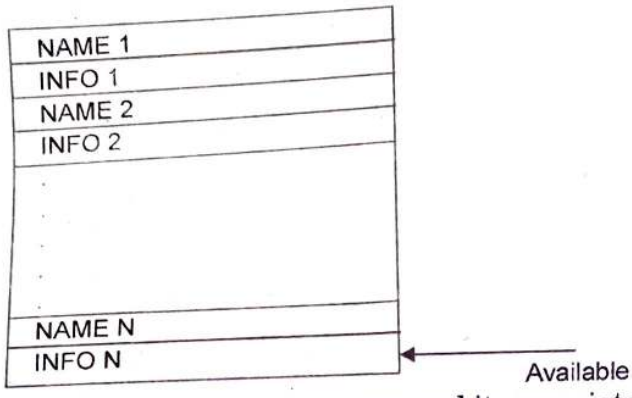
- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

Different data structures used for symbol table are:

1. Lists
2. Self Organising Lists
3. Trees
4. Hash Tables

LISTS:

1. It is simple and easy to implement data structure for symbol table in linear list records:



2. One can use single array to store the name and its associated information.
3. New names are added to end of list on FCFS basis.
4. Method of access is sequential

Advantages:

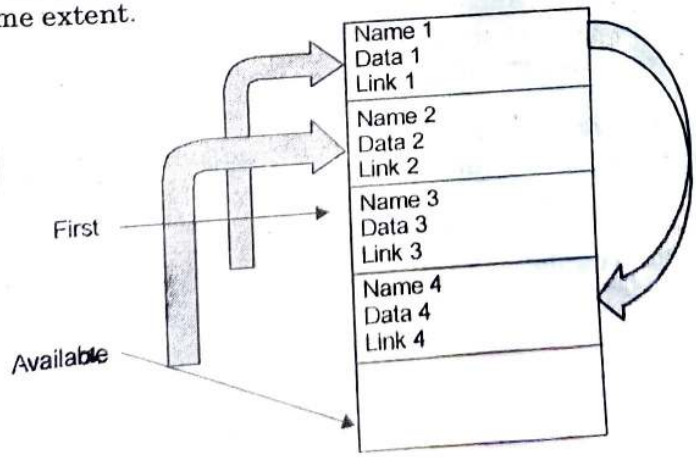
1. Minimum soace requirement.
2. Easy to understand and implement.

Disadvantages:

1. Sequential Access.
2. Amount of work done required to be done is high.

SELF ORGANISING LISTS:

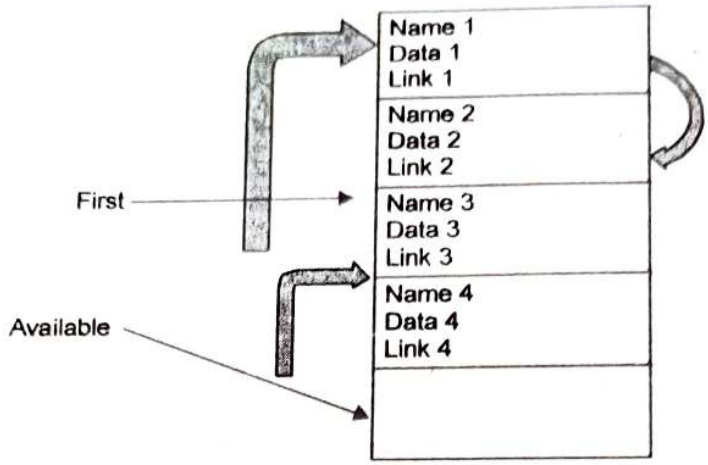
It is used to decrease searching time in list organisation.
Special attribute link is added to information of name that allows dynamic in list, this feature help in minimizing wastage of space and also promotes ility to some extent.



– First pointer points to the beginning of symbol table
– If we have names in the following order

3 -> 1 -> 4 -> 2

– If we need a particular record we need to move record to the front of the list. Suppose we need 4, so new connection will be



Mechanism

(*a*) Initial stage 3 -> 1 -> 4 -> 2

(*b*) 4th name is required

(*c*) move 4th to front

3 -> 1 -> **4** -> 2

(*d*) **4** -> 3 -> 1 -> 2

Advantages:

1. Increase search efficiency.
2. Promotes Reusability to some extent.

Disadvantage:

1. Difficult to maintain so many links and connection.
2. Extra space requirement is there.

HASH TABLES:

1. Hashing is important technique used to search the records of symbol table.
2. In hashing two tables are maintained.

(*a*) Hash table

(*b*) Symbol table

3. It consist of K entries from 0 to K-1. These entries are basically pointer to symbol table pointing to names of symbol table.

4. To determine whether "name" is in symbol table we, use hash function 'h' such that (Name) will result integer between 0 to K-1 we can search any name by position = h(Name).

5. Using this position we can obtain the exact location of name in symbol table.

6. Hash function should result in distribution of name in symbol table.

7. There should be minimum no. of collisions.Collision is a situation where hash function result in same location for storing names.

8.Various collision resolution techniques are Open Addressing, Chaining, Rehashing.

Advantages:

1. Quick Search

Disadvantage:

1. It is complicated to implement.

2. Extra space is required
3. Obtaining Scope of variables is difficult.

Types of Hash Function:

(1) **Mid Square Method:**

(*a*) Consider a key K

Let K = 3230

(*b*) Find the square of K

$K^2 = (3230)^2 = 10432900$

(*c*) Apply hash function

(*i*) Hash function will eliminate few digits from both sides of k2.

(*ii*) Hence H(K) = 32

Similarly

K	7236	2435
K2	52359696	5929225
H(K)	59	29

(*d*) Deleting the number of digits from both sides depend on maximum no. of keys quired, deletion may or may not be same from both ends.

(2) Division Method

Let Key = K

No. of elements = n

Hash function = H(K)

H(K) = K mod m or H(K) = K mod m + 1

here m = Prime no.

x: we have to index 100 records and hence largest prime number close to 100 is 97 ce we will choose m = 97

Let K = 3230

H(3230) = 3230 mod 97 = 29

(3) Folding Method

(*i*) Use this method for large key values.

(*ii*) We can use two methods to get hash value.

- Shift Folding

  K = 324 987 626 191 532

  Divide this key value in to parts K1, K2, K3, K4, K5

  K1 = 327

  K2 = 987

  K3 = 626

  = 191

  = 532

  2663

- Boundary Folding

  In this some of sub keys are reversed to generate new simple hash values

  K = 324 987 626 191 532

  Divide this key value in to parts K1, K2, K3, K4, K5

  K1 = 327

  K2 = 789

  K3 = 626

  K4 = 191

  K5 = 253

  2168

Ans. We know that arrays, records and other multi-word objects could be allocated either statically at the ends of it at the heap. The three kinds of allocation can be summarized on what they actually mean.

1. **Static allocation** Static allocation means that the data is allocated at a place in memory that has both known size and address at compile time. Furthermore, the allocated memory stays allocated throughout the execution of the program.

Most modern computers divide their logical address space into a text section used for code and a data section used for data. Assemblers programs that convert symbolic machine code into actual machine code usually maintain "current address" pointers to both the text area and the data area. They also have pseudo instructions (directives) that set these pointers to these addresses and move them. So you can allocate space for an array in the data space by placing a label at the current-address pointer in the data area and then move the current-address pointer up by the size of the array. The code can use the label to access the array. Allocation of space for an array A of 1000 integers (4 bytes each) can look like this in symbolic code:

```
A:  .space 4000  # declare 4000 byte array A
    .text        # switch back to text area for code generation
```

The base address of the array A is at the label name A.

The assembler (usually assisted by a linker) translates the symbolic code to binary code where references to names are replaced by references to numeric addresses. In the programming language C all global variables, regardless of size, are statically allocated.

Properties:

- The size of an array must be known at compile time if it is statically allocated, and the size can not change during execution.
- Furthermore the space is allocated throughout the entire execution of the program, even if the array is only in use for a small fraction of this time.
- In effect there is little sense of statically allocated space within one program unit. The programming language C++ at least allows the programmer to specify that

several statically allocated arrays can share the same space, but this feature is rarely seen in modern languages.

• It is, in theory, possible to analyze the liveness of arrays in the same way that we analyzed liveness of local variables , and use this to define interference between arrays in such a way that two arrays that do not interfere can share the same space. This is, however, rarely done, as it typically requires an expensive analysis of the entire program.

2. Stack allocation:

• The call stack can also be used to allocate arrays and other data structures. This is done by making room in the current (topmost) frame on the call stack. This is done by moving the frame pointer and/or the stack pointer.

Stack allocation has both advantages and disadvantages:

• The space for the array is freed up when the function that allocated the array returns (as the frame in which the array is allocated is taken off the stack).This allows easy reuse of the memory for later stack allocations in the same program execution.

• Allocation is fairly quick: You just move the stack pointer or the frame pointer. Releasing the memory again is also fast – again, you only move a pointer.

• The size of the array need not be known at compile time: The frame is at runtime created to be large enough to hold the array. If the size of the topmost frame can be extended after it is created, you can even stack-allocate arrays during execution of the function body.

• An unbounded number of arrays can be allocated at runtime, since recursive functions can allocate an array in each invocation of the function.

3. Heap allocation:

• The array will not survive return from the function in which it is allocated, so it can be used only locally inside this function and functions that it calls.

• Once allocated, the array can not be extended, as you can not (easily) "squeeze" more space into the middle of the stack, where the array is allocated, nor can you reallocate it at the top of the stack (unless it is the same frame), as the reallocated array will be released earlier than the original array.

In C, arrays that are declared locally in a function are stack allocated (unless declared static, in which case they are statically allocated). C allows you to return pointers to stack-allocated arrays and it is up to the programmer to make sure he never follows a pointer to an array that is no longer allocated. This often goes wrong. Other languages (like Pascal) avoid the problem by not allowing pointers to stack-allocated data to be returned from a function.

The limitations of static allocation and stack allocation are often too restricting: You might want arrays that can be resized or which survive the function invocation in which they are allocated. Hence, most modern languages allow heap allocation, which is also called dynamic memory allocation.

Data that is heap allocated stays allocated until the program execution ends, until the data is explicitly deallocated by a program statement or until the data is deemed dead by a run-time memory-management system, which then deallocates it. The size of the array (or other data) need not be known until it is allocated, and if the size needs to be increased later, a new array can be allocated and references to the old array can be redirected to point to the new array. The latter requires that all references to the old array can be tracked down and updated, but that can be arranged, for example, by letting all references to an array go through an indirection node that is never moved. Languages that use heap allocation can be classified by how data is deallocated:

Explicitly by commands in the program or automatically by the run-time memory management system. The first is called "manual memory management" and the latter "automatic memory management" or "automatic garbage collection".

Q. 7. (a) What is lexical phase, syntactic and semantic phase errors? (7.5)

Ans. Lexical phase errors: Lexical Phase breaks the program in to tokens i.e it recognizes the tokens, in program.

1. If after some processing, the lexical Analyzer discovers that no prefix of the remaining input belongs to any token class, then lexical errors are generated.
2. It is mainly generated because next token to be read in source program is misspelled.

Eg: Integer Constant out of bound(0 to 32767)

Syntactic Phase Errors: These errors occur when stream of token not follow the grammatical rules of programming language.

1. Missing ; at the end of the statement.
2. Missing Right Parenthesis (a +(b+c)
3. Misspelled keyword
4. Colon in place of semicolon ex: I = 1:

Semantic phase errors

1. Non Declared variables
2. Type conflict of operands
3. Incorrect procedure calls (e.g.Wrong no. of Parameters).

Q. 7. (b) Discuss storage mechanism in block structured and non-block structured languages. (5)

Ans. Compiler must carry out the storage allocation and provide access to variables and data. Allocation can be done in two ways:

Compiler Time Allocation or Static Storage Allocation: A static storage allocation decision is taken by the compiler by looking only at the text of the program, but not by looking at what the program does when it executes. Allocation is done at compile time

Scope storage allocation includes,

- Definition of procedure
- Declaration of a name or variable
- Scope of declaration

Runtime Allocation or Dynamic Storage Allocation: A storage allocation decision is dynamic if it can be decided only while program is running. Allocation is done at runtime .Storage allocation can be done for two types of data variable:

1. Local data
2. Non local data

The local data can be handled using activation record where as non-local data can be handled using scope information. The block structured storage allocation can be done using static scope. The non-block structured allocation can be done using dynamic scope.

Dynamic storage allocation includes,

- Activation of procedure
- Binding of a name
- Lifetime of a binding

Activation Record

- Information needed for each execution of a procedure is stored in a record called activation record

• Procedure calls and returns are usually managed by a runtime stack called the control stack

• Each live activation has an activation record or a frame

• The root of activation tree is a the bottom of the stack

• The current execution path specifies the content of the stack with the last activation as record in the top of the stack

• Activation record contains 7 fields which are:

Returned
Actual parameters
Control link
Access link
Machine status
Local data
Temporary data

Size of each field of activation record can be estimated at compile time.

UNIT-IV

Q. 8. (a) What are the main steps in local optimizations? Optimize the following three address code and analyze how this is reducing execution time. (7.5)

PROD=0
1=1
T 1=4*1
T2=addr(A)-4
T3=T2[T1]
T4=addr(B)-4
T5=T4[T1]
T6=T3*T5
PROD=PROD+T6
1 = 1+1
If 1<=30 goto (stmt 3)

Ans.

(1) T1 = 4 * 1
(2) T2 = addr(A) – 4
(3) T3 = T2[T1]
(4) T5 = addr(B) – 4
(5) T6 = T5[T1]
(6) T7 = T3 * T6
(7) PROD = PROD + S7
(8) I = I + 1
(9) If I < = 30, goto (1)

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed. A code optimizing process must follow the three rules given below:

• The output code must not, in any way, change the meaning of the program.

• Optimization should increase the speed of the program and if possible, the program should demand less number of resources.

Optimization should itself be fast and should not delay the overall compiling process.

Q. 8. b) Explain peephole optimization. **(7.5)**

Ans. Peephole optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window". It works by recognising sets of instructions that can be replaced by shorter or faster sets of instructions. Common techniques applied in peephole optimization:

• Constant folding – Evaluate constant subexpressions in advance.
• Strength reduction – Replace slow operations with faster equivalents.
• Null sequences – Delete useless operations.
• Combine operations – Replace several operations with one equivalent.
• Algebraic laws – Use algebraic laws to simplify or reorder instructions.
• Special case instructions – Use instructions designed for special operand cases.
• Address mode operations – Use address modes to simplify code.

Redundant instruction elimination

At source code level, the following can be done by the user:

```
int add_ten(int x)
{
int y, z;
y = 10;
z = x + y;
return z;
}
```

```
int add_ten(int x)
{
int y;
y = 10;
y = x + y;
return y;
}
```

```
int add_ten(int x)
{
int y = 10;
return x + y;
}
```

```
int add_ten(int x)
{
return x + 10;
}
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

• MOV x, R0
• MOV R0, R0

We can delete the first instruction and re-write the sentence as:

MOV x, R0

Unreachable code

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

Example:

```
void add_ten(int x)
{
return x + 10;
printf("value of x is %d", x);
}
```

In this code segment, the printf statement will never be executed as the program control returns back before it can execute, hence printf can be removed.

Flow of control optimization

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
MOV R1, R2
GOTO
...
L1 : G
L2 : I
In this
jumping to
...
MOV I
GOTO
...
L2 : I
```

Algebr

There
the expres
simply be

Streng

There
reduced by
produce th

For ex
Though th

Acces

The ta
capability
accommod
but also y

Q. 9.

T_8=D
T_6=A
T_5=T
T_4=T
T_3=T
T_2=T
T_1=T

Ans.

Algebraic expression simplification

There are common algebraic expressions that can be made simple. For example, the expression $x = x + 0$ can be replaced by x itself and the expression $x = x * 1$ can simply be replaced by x.

Strength reduction

There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result.

For example, $x * 2$ can be replaced by $x + x$, which involves only one left shift. Though the output of $x * x$ and x^2 is same, x^2 is much more efficient to implement.

Accessing machine instructions

The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently. If the target code can accommodate these instructions directly, that will not only improve the quality of code, but also yield more efficient result.

Q. 5. (a) What is DAG? Draw the DAG for the following statement: (7.5)

$T_0 = B + E$

$T_1 = A + B$

$T_2 = T_0 - C$

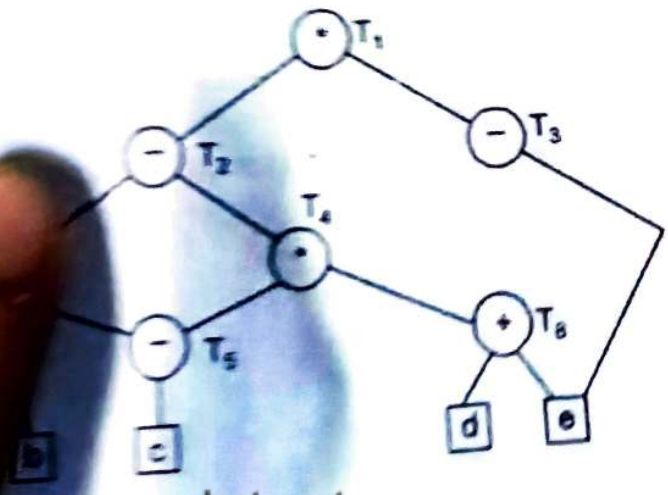
$T_3 = T_2 * T_1$

$T_4 = T_3 - E$

$T_5 = T_0 - T_4$

$T_6 = T_4 * T_5$

Ans.



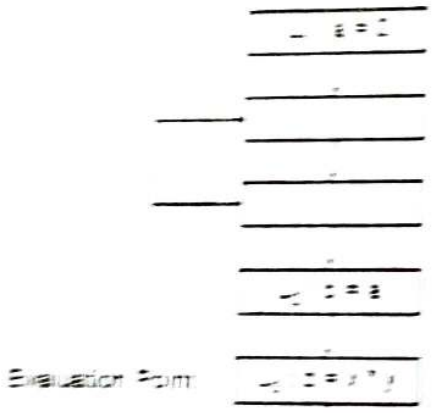
A directed acyclic graph DAG is a graph that is directed and without cycles connecting the other edges. This means that it is impossible to traverse the entire graph starting at one edge. The edges of the directed graph only go one way. The graph is a topological ordering where each node is in a certain order. A directed acyclic graph means that the graph is not cyclic or that it is impossible to start at one point in the graph and traverse the entire graph. Each edge is directed from an earlier edge to a later edge.

Q. 5. (a) Explain data flow analysis with reaching definitions. (5)

Ans. Data Flow Analysis It is the analysis of flow of data in control flow graph, i.e. the analysis that determines the information regarding the definition and use of data in program. With the help of this analysis optimization can be done.

Basic Terminologies –

- **Definition Point:** A point in a program containing some definition.
- **Reference Point:** A point in a program containing a reference to a data item.
- **Evaluation Point:** A point in a program containing evaluation of expression.



Data Flow Properties –

- **Available Expression** – A expression is said to be **available at a program point x** if along paths its reaching to x. A Expression is **available at its evaluation point.** A expression is said to be available if none of **the operands gets modified before** their use. It is used to eliminate common sub expressions.
- **Reaching Definition** – A definition D is **reaches a point x if there is path from D** to x if it is not killed, i.e., not redefined. **It is used in constant and variable propagation.**
- **Live variable** – A variable is said to be **live at some point p if from p to end the** variable is used before it is redefined else it **becomes dead. It is useful for register** allocation and it is used in dead code elimination.
- **Busy Expression** – An expression is **busy along a path iff its evaluation exists** along that path and none of its operands definition **exists before its evaluation along the** path. It is used for performing code movement optimization.