

CS8602 COMPILER DESIGN

CS8602 COMPILER DESIGN**OBJECTIVES:**

- To learn the various phases of compiler.
- To learn the various parsing techniques.
- To understand intermediate code generation and run-time environment.
- To learn to implement front-end of the compiler.
- To learn to implement code generator.

UNIT I INTRODUCTION TO COMPILERS

Structure of a compiler – Lexical Analysis – Role of Lexical Analyzer – Input Buffering – Specification of Tokens – Recognition of Tokens – Lex – Finite Automata – Regular Expressions to Automata – Minimizing DFA.

UNIT II SYNTAX ANALYSIS

Role of Parser – Grammars – Error Handling – Context-free grammars – Writing a grammar – Top Down Parsing - General Strategies Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item Construction of SLR Parsing Table - Introduction to LALR Parser - Error Handling and Recovery in Syntax Analyzer-YACC.

UNIT III INTERMEDIATE CODE GENERATION

Syntax Directed Definitions, Evaluation Orders for Syntax Directed Definitions, Intermediate Languages: Syntax Tree, Three Address Code, Types and Declarations, Translation of Expressions, Type Checking.

UNIT IV RUN-TIME ENVIRONMENT AND CODE GENERATION

Storage Organization, Stack Allocation Space, Access to Non-local Data on the Stack, Heap Management - Issues in Code Generation - Design of a simple Code Generator.

UNIT V CODE OPTIMIZATION

Principal Sources of Optimization – Peep-hole optimization - DAG- Optimization of Basic Blocks-Global Data Flow Analysis - Efficient Data Flow Algorithm.

COURSE OUTCOMES:

At the end of the course, the student should be able to:

CO1	Understand the different phases of compiler.
CO2	Design a lexical analyzer for a sample language.
CO3	Apply different parsing algorithms to develop the parsers for a given grammar.
CO4	Understand syntax-directed translation and run-time environment.
CO5	Learn to implement code optimization techniques and a simple code generator.
CO6	Design and implement a scanner and a parser using LEX and YACC tools.

UNIT I INTRODUCTION TO COMPILERS**PART – A****1. What is a compiler? (R) (May/June 2008) (May/June 2012)**

A compiler is a program that reads a program written in one language –the source language and translates it into an equivalent program in another language-the target language. The compiler reports to its user the presence of errors in the source program.

2. What are the two parts of a compilation? (R)(May/June 2009) (May/June 2016) (May/June 2017)(APR/MAY 2017)(Nov/Dec 2018) (NOV/DEC 21)**What are the two parts of a compilation and its function?(Apr/May 2018)**

Analysis and Synthesis are the two parts of compilation. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. It includes Lexical (Linear) Analysis, Syntax (Hierarchical) Analysis, and Semantic Analysis. The synthesis part constructs the desired target program from the intermediate representation. It includes Code optimization phase, Code generation.

3. List the subparts or phases of analysis part. (R)**List out the phases included in the analysis phase of compiler.(APR/MAY 22)**

Analysis consists of three phases:

- Linear Analysis or Lexical Analysis
- Hierarchical Analysis or parsing or syntax analysis
- Semantic Analysis
- Intermediate code generator

4. What is linear analysis? (R)

Linear analysis is one in which the stream of characters making up the source program is read from left to right and grouped into tokens that are sequences of characters having a collective meaning. Also called lexical analysis or scanning.

5. List the various phases of a compiler. (R) (Nov/Dec 2008)

The following are the various phases of a compiler:

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermediate code generator
- Code optimizer
- Code generator

6. What are the classifications of a compiler? (R)

Compilers are classified as:

- Single- pass
- Multi-pass
- Load-and-go
- Debugging or optimizing

7. What is a symbol table? (R)

A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. Whenever an identifier is detected by a lexical analyzer, it is entered into the symbol table. The attributes of an identifier cannot be determined by the lexical analyzer.

8. State some software tools that manipulate source program? (U) (May/June 2013)

Structure editors
Pretty printers
Static checkers
Interpreters

9. Mention some of the cousins of a compiler. (R) (April/May 2004, April/May 2005) (APRIL/MAY 2017) (NOV/DEC 2021)

Cousins of the compiler are:
Preprocessors
Assemblers
Loaders and Link-Editors

10. List the phases that constitute the front end of a compiler. (R) (May/June 2013)

The front end consists of those phases or parts of phases that depends primarily on the source language and are largely independent of the target machine. These include

- Lexical and Syntactic analysis
- The creation of symbol table
- Semantic analysis
- Generation of intermediate code

A certain amount of code optimization can be done by the front end as well. Also includes error handling that goes along with each of these phases.

11. What is a Structure editor? (R)

A structure editor takes as input a sequence of commands to build a source program .The structure editor not only performs the text creation and modification functions of an ordinary text.

12. Mention the back-end phases of a compiler. (R)

The back end of the compiler includes those portions that depend on the target machine and generally those portions do not depend on the source language, just the intermediate language. These include

- Code optimization
- Code generation, along with error handling and symbol- table operations.

13. Define compiler-compiler. (R)

Systems to help with the compiler-writing process have often been referred to as compiler-compilers, compiler-generators or translator-writing systems. Largely they are oriented around a particular model of languages, and they are suitable for generating compilers of languages similar model.

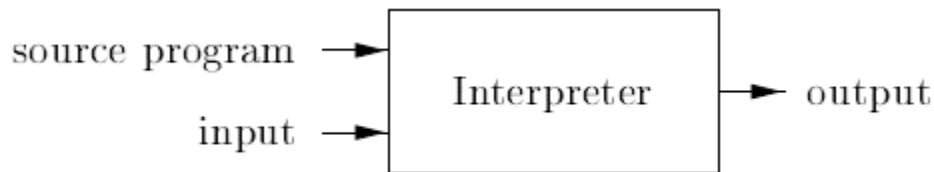
14. List the various compiler construction tools. (R) (April /May 2008) (April/May 2011) (Nov/Dec 2016)

The following is a list of some compiler construction tools:

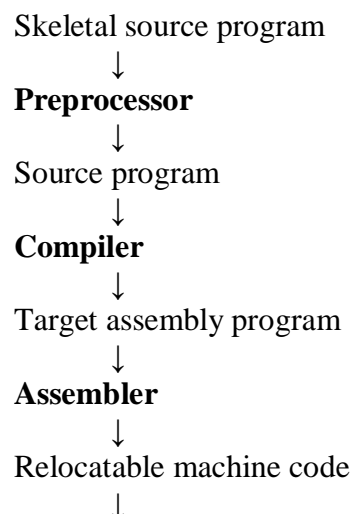
- Parser generators
- Scanner generators
- Syntax-directed translation engines
- Automatic code generators
- Data-flow engines

15. What is an interpreter? (R) (Nov/Dec 2017)

An interpreter is a translator which translates the source program line by line and executes the operations specified in the source program on inputs supplied by the user.

**16. Illustrate diagrammatically how a language is processed(AN) (May/June 2016)**

How source code is translated to machine code? (Nov/Dec 2018)



Loader/ link editor ←library, relocatable object files
↓
Absolute machine code

17. State any two reasons as to why phases of compiler should be grouped. (May/June 2014)How will you group the phases of compiler?(R)(Nov/Dec 2013)

A Compiler operates in phases,each of which transforms the source program from one representation into another representation.They communicate with error handlers.Logically each phase is viewed as a separate program that reads input and produces output for the next phase (i.e.) a pipeline.

18. What are the functions of preprocessors? (R) (May/June 2009)

Program that processes the source code before the compiler sees it. Usually, it implements macro expansion, but it can do much more.

The functions of a preprocessor are:

- Macro processing
- File inclusion.
- Rational preprocessors
- Language extensions

19. Distinguish between compiler and interpreter(AN) (Nov/Dec 2008)

Compiler is a program that reads a program written in one language –the sourceLanguage and translates it into an equivalent program in another language- the targetlanguage. In this translation process, the compiler reports to its user the presence of the errors in the source program.

Interpreter is a language processor program that translates and executes source code directly, without compiling it to machine code.Compiler produces a target program whereas an interpreter performs the operations implied by the source program.

20. Give examples for static check. (U) (May/June 2013)

It may detect the parts of a source program that can never be executed

It can catch logical errors.

21. Define Cross Compilers (R)(NOV/DEC 2017)(NOV/DEC 2021)

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.

22. Describe the possible error recovery actions in lexical analyzer.(U)(APR/MAY 2018)

1. Deleting an extraneous character
2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. Transposing two adjacent characters

**23. Differentiate tokens, patterns, lexeme. (AN) (May /June 2013, Nov/Dec 2010)
(Nov/Dec 2016)(NOV/DEC 2017) (NOV/DEC 2021)**

Discriminate tokens, patterns and lexemes.(April/May 2019)

Tokens- Sequence of characters that have a collective meaning.

Patterns- There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme- A sequence of characters in the source program that is matched by the pattern for a token.

24. List the operations on languages. (R) (May/June 2016)

- Union - $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
- Concatenation – $LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
- Kleene Closure – L^* (zero or more concatenations of L)
- Positive Closure – L^+ (one or more concatenations of L)

25. Write a regular expression for an identifier and number. (C) (April/May 2017)

An identifier is defined as a letter followed by zero or more letters or digits.

The regular expression for an identifier is given as

letter (letter | digit)*

26. Mention the various notational shorthands for representing regular expressions.(R)

- One or more instances (+)
- Zero or one instance (?)
- Character classes ([abc] where a,b,c are alphabet symbols denotes the regular expressions $a \mid b \mid c$.)
- Non regular sets

27. List the various error recovery strategies for a lexical analysis. (R)

(Nov/Dec 2008)(Nov/Dec 2015) (NOV/DEC 2021)

Possible error recovery actions are:

- Panic mode recovery
- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by a correct character
- Transposing two adjacent characters

28. Why lexical and syntax analyzers are separated out? (AN)

Reasons for separating the analysis phase into lexical and syntax analyzers:

Simpler design.

- Compiler efficiency is improved.
- Compiler portability is enhanced.

29. What is a lexeme? Define a regular set. (R) (Nov/Dec 2006)(Nov/Dec 2010)

A Lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

A language denoted by a regular expression is said to be a regular set.

30. What is a sentinel? What is its usage? (R) (April/May 2004)(Nov/Dec 2010)

A Sentinel is a special character that cannot be part of the source program. Normally use 'eof' as the sentinel. This is used for speeding-up the lexical analyzer

31. What is Lexical Analysis?Mention the issues in a lexical analyzer. (R) (May /June 2013)

The first phase of compiler is Lexical Analysis. This is also known as linear analysis in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

Issues:

Lexical analyzer doesn't return a list of tokens at one short, it returns a token when the parser asks a token from it. There are several reason for separating the analysis phase of compiling into lexical analysis and parsing:

- Simpler design is perhaps the most important consid
- eration. The separation of lexical
- analysis often allows us to simplify one or other of these phases.
- Compiler efficiency is improved.
- Compiler portability is enhanced.

32. What are the Error-recovery actions in a lexical analyzer? (R) (May/June 2013)

1. Deleting an extraneous character
2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. Transposing two adjacent characters

33. Write a short note on LEX. (U)

A LEX source program is a specification of lexical analyzer consisting of set of regular expressions together with an action for each regular expression. The action is a piece of code, which is to be executed whenever a token specified by the corresponding regular expression is recognized. The output of a LEX is a lexical analyzer program constructed from the LEX source specification.

34. What are the components of Lex? (R) (NOV/DEC 2015)

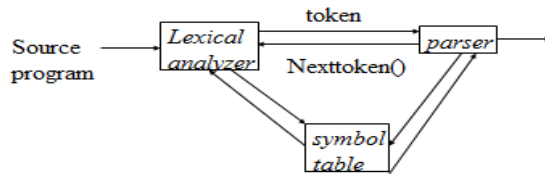
Declarations
Translation rules
Auxiliary Procedures

35. Why buffering used in lexical analysis? What are the commonly used buffering methods? (AN) (May/June 2014)

Buffering techniques have been developed to reduce the amount of overhead required to process a single input character.

36. What is phrase level error recovery? (R) (Nov/Dec 2008)

Phrase level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack.

37. State the interactions between the lexical analyzer and the parser. (AN) (Nov/Dec 2015) (Nov/Dec 2021)**38. What are the possible error-recovery actions in lexical Analyzer? (R) (May/June 2013)**

Possible error recovery actions are:

- Panic mode recovery
- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by a correct character
- Transposing two adjacent characters

39. Define Lexeme. (R) (May/June 2014) (Nov/Dec 2017)

A **Lexeme** is a sequence of characters in the source program that is matched by the pattern for a token. A language denoted by a regular expression is said to be a regular set.

40. Compare the features of DFA and NFA. (AN) (May/June 2014)

1. DFA stands for "Deterministic Finite Automata" while "NFA" stands for "Nondeterministic Finite Automata."
2. Both are transition functions of automata. In DFA the next possible state is distinctly set while in NFA each pair of state and input symbol can have many possible next states.
3. NFA can use empty string transition while DFA cannot use empty string transition.
4. NFA is easier to construct while it is more difficult to construct DFA.
5. Backtracking is allowed in DFA while in NFA it may or may not be allowed.
6. DFA requires more space while NFA requires less space.
7. While DFA can be understood as one machine and a DFA machine can be constructed for every input and output, 8. NFA can be understood as several little machines that compute together, and there is no possibility of constructing an NFA machine for every input and output.

41. Write a regular expression to describe a language consist of strings made of even numbers a and b. (C) (Nov/Dec 2014)

$(aa | bb)^* ((ab | ba) (aa | bb)^* (ab | ba) (aa | bb)^*)^*$

42. State the kinds of data that appear in activation record? (U) (Nov/Dec 2015)

Local data :is a data that is local to the execution of procedure is stored in this field of activation record.

43. What does a semantic analysis do?(R)

Semantic analysis is one in which certain checks are performed to ensure that components of a program fit together meaningfully. Mainly performs type checking

**44. What is a regular expression? State the rules, which define regular expression?(R)
Apply the rules used to define a regular expression. Give Example. (Apr/May 2018)
State the rules to define regular expression. (Nov/Dec 2018)**

Regular expression is a method to describe regular language

- 1) If a is a symbol in Σ , then a is a regular expression that denotes $\{a\}$
- 2) Suppose r and s are regular expressions denoting the languages $L(r)$ and $L(s)$

Then,

- a) $(r)(s)$ is a regular expression denoting $L(r)U L(s)$.
- b) $(r)(s)$ is a regular expression denoting $L(r)L(s)$
- c) $(r)^*$ is a regular expression denoting $L(r)^*$.
- d) (r) is a regular expression denoting $L(r)$.

45. What are the Error-recovery actions in a lexical analyzer?(R)

1. Deleting an extraneous character
2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. Transposing two adjacent characters

46. Construct Regular expression for the language(C)(Nov/Dec 2018)

$L = \{w \in \{a,b\}^* / w \text{ ends in } abb\}$

Ans: $\{a/b\}^*abb$.

47. What are the various parts in LEX Program? (Apr/May 2017)

DECLARATIONS

%%

RULES

%%

AUXILIARY FUNCTIONS

48. List the rules that form the BASIS.(NOV/DEC 2021)

There are two rules that form the basis:

1. ϵ is a regular expression and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.

2. If a is a symbol in Σ , then a is a regular expression, and $L(a) = \{a\}$, that is, the language with one string, of length one, with a in its one position.

49. What is the role of lexical analyzer phase?(R) (NOV/DEC 2017)

The main role of lexical analyzer is to read the input characters of the source program, and produce as output a sequence of tokens that the parser uses for syntax analysis. The lexical analyzer is implemented as a sub-routine or co-routine of the parser.

50. Differentiate NFA and DFA.(AN) (NOV/DEC 2017)

NFA	DFA
Deterministic Finite Automaton is a FA in which there is only one path for a specific input from current state to next state. There is a unique transition on each input symbol.	NFA or Non Deterministic Finite Automaton is the one in which there exists many paths for a specific input from current state to next state.
DFA cannot use Empty String transition	NFA can use Empty String transition.
DFA can be understood as one machine	NFA can be understood as multiple little machines computing at the same time.
DFA will reject the string if it end at other than accepting state	If all of the branches of NFA dies or rejects the string, we can say that NFA reject the string.
For Every symbol of the alphabet, there is only one state transition in DFA.	We do not need to specify how the NFA reacts according to some symbol.

51. List the attributes stored in symbol table.

Variable names and constants
 Procedure and function names
 Literal constants and strings
 Compiler generated temporaries
 Labels in source languages

52. Why is compiler optimization essential?

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result.

53. Write the regular expression for all valid identifiers.

Identifier: $[_a-zA-Z][_a-zA-Z0-9]^*$

54. Programmer A has written a program which needs to be modified very frequently. Which of the 2 languages Visual Basic (or) C++ can he use for his programming? Justify in 2 sentences. (Neglect other technical and environmental considerations) (APRIL/MAY 2022)

Python is preferable. It uses interpreter for converting source program to object program. Since, programmer A has written a program it needs to be modified very frequently, interpreter, that converts the source program line by line is preferable.

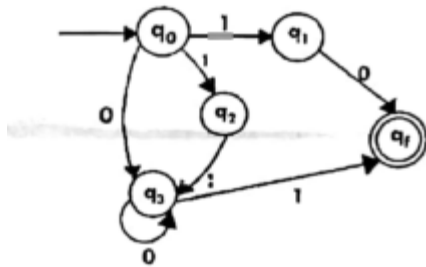
PART – B

1. Describe the following software tools (U)(April/May 2011)
 - i. Structure Editors
 - ii. Pretty printers
 - iii. Interpreters
2. Write in detail about the cousins of the compiler. (C)(May/June 2008)(May/June 2013,Nov/Dec 2013)
3. Explain the various phases of compiler in detail, with a neat sketch. (U)(April/May 2008) (May/June 2012)(Nov/Dec 2016)(NOV/DEC 2017)(APR/MAY 2018)(NOV/DEC 2021)
4. Elaborate on grouping of phases in a compiler. (U)(May/June 2013) (May/June 2008)
5. Explain the various phases of a compiler in detail. Also write down the output of the following expression after each phase $a := b * c - d$. (U)(May/June 2009)
6. What are the phases of the compiler? Explain the phases in detail. Write down the output of each phase for the expression $a = b * c + 50 - d$. (U)(May/June 2013)
7. Explain various phases of compiler in detail. Write the output of each phase of the compiler for the expression $C := a + b * 12$ (U) (May/June 2019)
8. What are the phases of the compiler? Explain the phases in detail. Write down the output of each phase for the expression $a := b + c * 60$ (R)(April/May 2017)
9. Write a short note on front and back end of the compiler. (U)
10. Explain Symbol table management and error handling(R)
11. Explain compiler construction tools in the compiler.(OR)What are the characteristics of compiler construction tools?Explain how compiler construction tool help in implementation of various phases of a compiler. (R)(Nov/Dec 2014, Apr/May 2015) (April/May 2017) (Nov/Dec 2016)(Nov/Dec 2018)(April/May 2019) (NOV/DEC 2021)
12. Explain cousins of compiler (R)(Nov/Dec 2008) (April/May 2017)
13. Define the following terms: Compilers, Interpreter, Translator and differentiate between them (R)(May/June 2014)
14. Explain in detail the process of compilation. Illustrate the output of each phase of compilation for the input: $a = (b+c)*(b+c)*2$. (E)(May/June 2014) (Nov/Dec 2018)
15. Explain the need for grouping of phases of a compiler (R)(Nov/Dec2014)(Nov/Dec2016)(NOV/DEC 2021)
16. Explain in detail about language processing system.(R)
17. Analyze the given expressions $4 := cba$ with different phases of the compiler(AN)
18. Draw the transition diagram for relational operators and unsigned numbers. (C) (April/May 2017)(APR/MAY 2018)
19. Explain the various errors encountered in different phases of a compiler.(U)(Nov/Dec 2016)(NOV/DEC 2021)
20. What are compiler construction tools?Write note on each compiler construction tool.(U)(NOV/DEC 2017)
21. Draw a diagram for the compilation of a machine language processing system.(C) (APR/MAY 2018)
22. Apply the analysis phases of compiler for the following assignment statement.
Position:=initial+rate*60(AP) (APR/MAY 2018)

23. Outline the compiler construction tools can be used to implement various phases of a compiler. **(R)(APR/MAY 2018)**
24. Draw NFA for the regular expression ab^*/ab . **(C) (May/June 2014)**
25. Explain in detail about the role of Lexical analyzer with the possible error recovery actions. **(U)(May/June 2009,2013, Apr/May 2015)(Nov/Dec2011, 2018)**
26. Elaborate specification of tokens. **(U)(May/June 2013) (May/June 2008)**
27. Explain the role performed by lexical analysis of the compiler. **(U)(April/May 2011)(Nov/Dec 2016)(or) Analyze the role of lexical analyzer with suitable examples.(AN)(May/June 2019)**
28. Differentiate between lexeme, token and pattern. **(AN)(May/June 2014) (May/June 2016)(Apr/May 2017)(Nov/Dec 2018)**
29. What are the issues in Lexical Analysis? **(R) (May/June 2014)(May/June 2016)(Apr/May 2017) (NOV/DEC 2017)**
30. Write notes on regular expression. **(R)(May/June 2016)**
31. Prove that the following two regular expressions are equivalent by showing that the minimum state DFA's are same. **(E) (Apr/May 2015)**
 - i. $(a/b)^*$
 - ii. $(a^*/b^*)^*$
32. Explain specification and recognition of tokens. **(U) (Nov/Dec 2014) (Nov/Dec 2018)**
33. Write an algorithm for minimizing the number of states of a DFA. **(R)(Nov/Dec 2016)**
34. Write notes on regular expression to NFA. Construct Regular expression to NFA for the Sentence $(a|b)^*a$ **(A) (May/June 2016)**
35. Construct DFA to recognize the language $(a/b)^*ab$ **(A) (May/June 2016)**
36. Distinguish between context free grammar and regular expressions **(AN)(Nov/Dec 2015)**
37. State and explain the architecture of a transition – diagram- based lexical analyzer. **(U)(Nov/Dec 2015)**
38. How to minimize the number of states of DFA. Explain it with example. **(AN)(Nov/Dec 2015)**
39. Explain the procedure for construction of an NFA from a regular expression **(U) (Nov/Dec 2015)(NOV/DEC 2021)**
40. What are the functions computed from the syntax tree. Explain each function with example. **(R)(Nov/Dec 2015)**
41. Convert a Regular Expression $ab(a|b)^*$ to DFA using direct method and minimize it. **(C)(Apr/May 2017)**
42. Draw transition diagram for relational operators and unsigned numbers. **(C)(Apr/May 2017)**
43. Discuss how finite automata is used to represent tokens and perform lexical analysis with examples. **(U)(Nov/Dec 2016)**
44. Conversion of regular expression $(a/b)^*ab$ to NFA. **(U)(Nov/Dec 2016)**
45. Draw NFA for regular expression ab^*/ab . **(C) (NOV/DEC 2017)**
46. Write an algorithm to convert NFA to DFA and minimized DFA. Give an example. **(U) (NOV/DEC 2017)**

47. Considering the alphabet $\Sigma = \{0,1\}$. Construct a Non deterministic Finite Automata(NFA) using the Thompson construction that is able to recognize the sentences generated by the regular expression $(1^*01^*0)^*1^*$. (C) (APR/MAY 2018)
48. Illustrate how does LEX works? (U) (APR/MAY 2018)
49. Consider the regular expression below which can be used as part of a specification of the definition of exponents in floating point numbers. Assume that the alphabet consists of numeric digits ('0' through '9') and alphanumeric characters ('a' through 'z' and 'A' through 'Z') with the addition of a selected small set of punctuation and special characters. (say in this example only the characters '+' and '-' are relevant). Also, in this representation of regular expressions the character '.' Denotes concatenation.
 Exponent = $(+|-|\epsilon).(E|e).(digit)^+$
- Derive an NFA capable of recognizing its language using Thompson construction.
 - Derive the DFA for the NFA found in a) above using the subset construction.
 - Minimize the DFA found in (ii) above using the interactive refinement algorithm described in class. (C) (APR/MAY 2018)
50. Explain the functions of the Lexical Analyzer with its implementation (U)
51. What are Lex and Lex specification? How lexical analyzer is constructed using lex? Write a Lex program that recognizer the tokens. (C)
52. Explain conversion of regular expression to DFA with an example. (U)
53. Explain in detail about converting a Regular Expression into a Deterministic Finite Automaton (U)
54. Construct the minimized DFA for the RE $(0+1)^*(0+1)01$.
55. Draw and explain the translation diagram that recognizes the lexemes matching the token relop (relational operator). (C) (APR/MAY 2019)
56. Write the subset construction algorithm. Using the subset construction algorithm, convert the regular expression $(a|b)^*abb$ to DFA.
57. In SQL, keywords and identifiers are case-insensitive. Write a Lex program that recognizes the keywords SELECT, FROM, and WHERE (in any combination of capital and lower-case letters), and token ID, which may be any sequence of letters and digits, beginning with a letter. (C) (APR/MAY 2019) (NOV/DEC 2021)
58. Explain the procedure for constructing a DFA from an NFA with example. (NOV/DEC 2021)
59. Draw the transition graph for an NFA that recognizes the language aa^*/bb^* . (NOV/DEC 2021)
60. How to minimize the number of states of DFA? Explain it with an example. (NOV/DEC 2021)
61. How a finite automaton is used to represent tokens and perform lexical analysis with examples. (NOV/DEC 2021)
62. Compare and Contrast NFA and DFA. (NOV/DEC 2021)
63. Elaborate on the different phases of a compiler with a neat sketch. Show the output of each phases of the compiler when the following statement is parsed.
 $SI = (p * n * r) / 100$
 Where, n should be an integer
 P and r could be a floating point numbers (APR/MAY 2022)

64. Convert the following NFA to DFA. (April/May 2022)



65. Find transition diagrams for the following regular expression and regular definition. (NOV/DEC 2021)

- $a(ab)^*a$
- $((|a)b^*)^*$
- All strings of digits with at most one repeated digit.
- All strings of a's and b's that do not contain the substring abb.
- All strings of a's and b's that do not contain the subsequence abb

UNIT II SYNTAX ANALYSIS

Part- A

1. Define parser. (R) (April/May 2011)

The parser obtains a string of tokens from the lexical Analyzer and verifies that the string of token names can be generated by the grammar for the source language.

The parsers reports any syntax errors and to recover from commonly occurring errors to continue processing the remainder of the program.

2. Mention the basic issues in parsing. (R)

There are two important issues in parsing.

- Specification of syntax
- Representation of input after parsing.

3. Define a context free grammar. (R) (May/June 2009)

A context free grammar G is a collection of the following

- V is a set of non terminals
- T is a set of terminals
- S is a start symbol
- P is a set of production rules

G can be represented as $G = (V, T, S, P)$

Production rules are given in the following form

Non terminal $\rightarrow (V \cup T)^*$

4. Briefly explain the concept of derivation. (U)

Derivation from S means generation of string w from S . For constructing derivation two things are important.

- i) Choice of non terminal from several others.
- ii) Choice of rule from production rules for corresponding non terminal. Instead of choosing the arbitrary non terminal one can choose either
leftmost derivation – leftmost non terminal in a sentinel form
or
rightmost derivation – rightmost non terminal in a sentinel form

5. Define ambiguous grammar. (R)(May/June 2016) (NOV/DEC 2021)

A grammar G is said to be ambiguous if it generates more than one parse tree for some sentence of language L(G).

i.e. both leftmost and rightmost derivations are same for the given sentence.

Demerit :

It is difficult to select or determine which parse tree is suitable for an input string

6. What is operator precedence parser? (R) (April/May 2011)

A grammar is said to be operator precedence if it possess the following properties:

- 1. No production on the right side is ϵ .
- 2. There should not be any production rule possessing two adjacent non terminals at the right hand side.

7. List the properties of LR parser. (R) (May/June 2012)

- 1. LR parsers can be constructed to recognize most of the programming languages for which the context free grammar can be written.
- 2. The class of grammar that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsers.
- 3. LR parsers work using non backtracking shift reduce technique yet it is efficient one.

8. Mention the types of LR parser. (R) (May/June 2012)

- SLR parser- simple LR parser
- LALR parser- lookahead LR parser
- Canonical LR parser

9. What are the problems with top down parsing? (R)

The following are the problems associated with top down parsing:

- Backtracking
- Left recursion
- Left factoring
- Ambiguity

10. Write the algorithm for FIRST and FOLLOW. (C)(May/June 2009) (April/May 2011) (May/June 2016)

FIRST

- 1. If X is terminal, then FIRST(X) IS {X}.
- 2. If $X \rightarrow e$ is a production, then add e to FIRST(X).
- 3. If X is non terminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then place a in FIRST(X) if

for some i , a is in $FIRST(Y_i)$, and e is in all of $FIRST(Y_1), \dots, FIRST(Y_{i-1})$;

FOLLOW

1. Place $\$$ in $FOLLOW(S)$, where S is the start symbol and $\$$ is the input right endmarker.
2. If there is a production $A \rightarrow aB\beta$, then everything in $FIRST(\beta)$ except for e is placed in $FOLLOW(B)$.
3. If there is a production $A \rightarrow aB$, or a production $A \rightarrow aB\beta$ where $FIRST(\beta)$ contains e , then everything in $FOLLOW(A)$ is in $FOLLOW(B)$.

11. List the advantages and disadvantages of operator precedence parsing.(R)

Advantages

This type of parsing is simple to implement.

Disadvantages

1. The operator like minus has two different precedence (unary and binary). Hence it is hard to handle tokens like minus sign.
2. This kind of parsing is applicable to only a small class of grammars.

12. What is dangling else problem? (R) (May/June 2012)

Ambiguity can be eliminated by means of dangling-else grammar which is shown below:

```
stmt  $\rightarrow$  if expr then stmt
      | if expr then stmt else stmt
      | other
```

13. Write short notes on YACC. (U)

YACC is an automatic tool for generating the parser program.

Basically YACC is an LALR parser generator. It can report conflict or ambiguities in the form of error messages.

14. What is meant by handle pruning?(U)(Nov/Dec 2016)((APR/MAY 2018, 2019)

A rightmost derivation in reverse can be obtained by handle pruning.

If w is a sentence of the grammar at hand, then $w \rightarrow^n$, where \rightarrow^n is the n th right-sentential form of some as yet unknown rightmost derivation

$$S \rightarrow^0 \Rightarrow \rightarrow^1 \dots \Rightarrow \rightarrow^{n-1} \Rightarrow \rightarrow^n = w$$

15. Define LR(0) items. (U) (May/June 2011))(Nov/Dec 2018)

An LR(0) item of a grammar G is a production of G with a dot at some position on the right side. Thus, production $A \rightarrow XYZ$ yields the four items

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$

16. What is meant by viable prefixes?(U)

The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes. An equivalent definition of a viable prefix is that it is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form.

17. Define handle. (U) (April/May 2011)

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.

A handle of a right – sentential form $\alpha \rightarrow \beta$ is a production $A \rightarrow \beta$ and a position of \rightarrow where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of α . That is, if $S \Rightarrow aAw \Rightarrow a\beta w$, then $A \rightarrow \beta$ in the position following a is a handle of $a\beta w$.

18. What are kernel & non-kernel items?(R) (NOV/DEWC 2021)

Kernel items, which include the initial item, $S' \rightarrow \cdot S$, and all items whose dots are not at the left end. On-kernel items, which have their dots at the left end.

19. What is phrase level error recovery? (R) (May/June 2010)

Phrase level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack.

20. What is a parse tree? (R) (Nov/Dec 2008)

A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding a replacement order. Each interior node of a parse tree is labeled by some nonterminal A and that the children of the node are labeled from left to right by symbols on the right side of the production by which this A was replaced in the derivation. The leaves of the parse tree are terminal symbols.

21. What are the disadvantages of operator precedence parsing? (R) (May/June 2007)

- (i) It is hard to handle tokens like the minus sign, which has two different precedences.
- (ii) Since the relationship between a grammar for the language being parsed and the operator – precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language.
- (iii) Only a small class of grammars can be parsed using operator precedence techniques.

22. Define left factoring. (U) (April/May 2011)(NOV/DEC 2021)

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal “ A ”, we may be able to rewrite the “ A ” productions to defer the decision until we have seen enough of the input to make the right choice.

23. What do you mean by viable prefixes? (U) (May/June 2012)

The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes. An equivalent definition of a viable prefix is that it is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form.

24. What are the goals of error handler in a parser? (U) (May/June 2010)

It should report the presence of errors clearly and accurately. It should recover from each error quickly. It should not significantly slow down the processing of correct programs.

25. What is phrase level error recovery?(R) (Nov/Dec 2008)

Phrase level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack.

26. Eliminate left recursion from the following grammar.(E) (April/May 2011)

$A \rightarrow Ac/Aad/bd/c.$

Solution

$A \rightarrow bdA'$

$A \rightarrow cA'$

$A' \rightarrow cA'$

$A' \rightarrow adA'$

$A' \rightarrow \epsilon$

27. What is LL (1) grammar? Give the properties of LL (1) grammar. (R)(May/June 2013)

LL(1) GRAMMARS AND LANGUAGES. A context-free grammar $G = (VT, VN, S, P)$ whose parsing table has no multiple entries is said to be LL(1). In the name LL(1), the first L stands for scanning the input from left to right, the second L stands for producing a leftmost derivation, and the 1 stands for using one input symbol of lookahead at each step to make parsing action decision. A language is said to be LL(1) if it can be generated by a LL(1) grammar. It can be shown that LL(1) grammars are not ambiguous and not left-recursive.

28. What is Left Factoring a Grammar?(R) (May/June 2009)

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

29. Define Lexeme. (U)(May/June 2014)

The character sequence forming a token is called lexeme for the token.

30. Eliminate left recursion from the following grammar.(AP) (April/May 2017)

$A \rightarrow Ac/Aad/bd.$

Solution:

$A \rightarrow bdA'$

$A' \rightarrow cA'|adA'|\epsilon$

31. What are the various conflicts that occur during shift reduce parsing?(R)(April/May 2017)

The entire stack contents and the next input symbol cannot decide whether to shift or reduce.

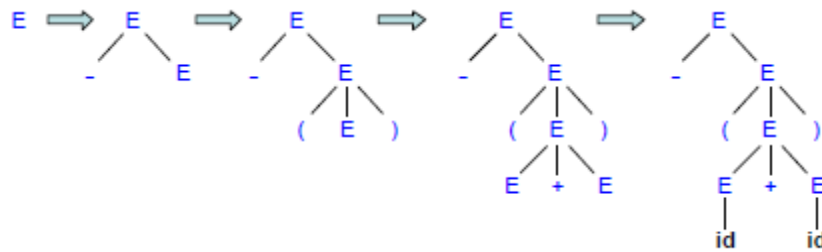
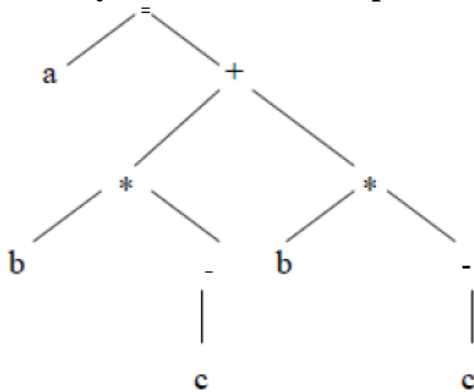
32. Construct a parse tree for $-(id + id)(C)$ (Nov/Dec 2017)

Given the following grammar

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

Lets examine this derivation:

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + id)$

**33. Draw syntax tree for the expression $a=b*-c+b*-c.(C)$ (NOV/DEC 2017)****34. Summarize merits and demerits of LALR parser.(AN) (Apr/May 2018)**

LALR is more or less a hack for LR parsers to make the tables smaller. The tables for an LR parser can typically grow enormous. LALR parsers give up the ability to parse all LR languages in exchange for smaller tables. Most LR parsers actually use LALR (not secretively though, you can usually find exactly what it implements).

LALR can complain about shift-reduce and reduce-reduce conflicts. This is caused by the table hack: it 'folds' similar entries together, which works because most entries are empty, but when they are not empty it generates a conflict. These kinds of errors are not natural, hard to understand and the fixes are usually fairly weird.

35. How do you identify predictive parser and non recursive predictive parser?(U) (APR/MAY 2018)(NOV/DEC 2021)

Predictive parser is top – down parsing. An efficient non-backtracking form of top-down parser called a predictive parser. LL(1) grammars from which predictive parsers can be constructed automatically.

Non recursive predictive parsing can be performed using a pushdown stack, avoiding recursive calls.

36. Write short notes on YACC. (or)

Mention the purpose of YACC.(May/June 2019)(U)

YACC is an automatic tool for generating the parser program.

YACC stands for Yet Another compiler which is basically the utility available for UNIX.

Basically YACC is LALR parser generator.

It can report conflict or ambiguities in the form of error messages.

**37. What are the different stages that a parser can recover from a syntactic error?(U)
(Nov/Dec 2018)**

- Panic mode
- Statement Mode recovery
- Error production
- Global Correction

38. Writedown the CFG for the set of odd length strings in {a,b}* whose first,middle and last symbols are same. (NOV/DEC 2021)

$S \rightarrow aSa \mid bSb$

$S \rightarrow aSa \mid aSb \mid bSa \mid bSb \mid \epsilon$

39. List out the steps for performing LR parsing.(NOV/DEC 2021)

1. Creation of augmented grammar G'
2. Create FIRST and FOLLOW for all the Nonterminals of G'
3. Create LR items of G'
4. Create LR parsing Table
5. Perform LR parsing

PART-B

1. What is the FIRST and FOLLOW? Explain in detail with an example. Write down the necessary algorithm. (C)(Nov/Dec 2008)
2. Construct Predictive Parsing table for the following grammar: (May/June 2013)

$S \rightarrow (L) / a$

$L \rightarrow L, S/S$

and check whether the following sentences belong to that grammar or not.

(i) (a,a)

(ii) (a, (a , a))

(iii) (a, ((a , a), (a , a))) (C)

3. Construct a predictive parsing table for the grammar

$S \rightarrow (L)a$

$L \rightarrow L,S|S$

and show whether the following string will be accepted or not.

(a,(a,(a, a))) .(Nov/Dec 2021)

4. Construct the predictive parser for the following grammar: (C)(Nov/Dec 2007)
(April/May 2017)

$S \rightarrow (L)a$

$L \rightarrow L,S|S.$

And show whether the following string will be accepted or not.(a,(a, (a , a)))

5. Construct the behaviour of the parser on sentence (a, a) using the grammar:
 $S \rightarrow (L)a$
 $L \rightarrow L, S | S$. **(A) (May/June 2009)**
6. Check whether the following grammar can be implemented using predictive parser. Check whether the string “abfg” is accepted or not using predictive parsing.
 $A \rightarrow A$
 $A \rightarrow aB | Ad$
 $B \rightarrow bBC | f$
 $C \rightarrow g$ **(April/May 2022)**
7. Check whether the following grammar can be implemented using predictive parser. Check whether the string “(a,a)” is accepted or not using predictive parsing.
 $S \rightarrow (L)a$
 $L \rightarrow L, S | S$ **(April/May 2022)**
8. For the grammar given below, calculate the operator precedence relation and the precedence functions. **(E) (May/June 2008) (May/June 2012)**
 $E \rightarrow E + E | E - E | E * E | E / E | E ^ E | (E) | -E | id$
9. Check whether the following grammar is a LL(1) grammar **(E)**
 $S \rightarrow iEtS | iEtSeS | a$
 $E \rightarrow b$
 Also define the FIRST and FOLLOW procedures. **(Nov/Dec 2007) (May/June 2009)**
10. Consider the grammar given below. **(May/June 2012, 2019)**
 $E \rightarrow E + T$
 $E \rightarrow T$
 $T \rightarrow T * F$
 $T \rightarrow F$
 $F \rightarrow (E)$
 $F \rightarrow id$.
 Construct an LR Parsing table for the above grammar. Give the moves of LR parser on $id * id + id$. **(C)**
11. What is a shift-reduce parser? Explain in detail the conflicts that may occur during shift-reduce parsing. **(R)**
12. Consider the following grammar:
 $S \rightarrow AS | b$
 $A \rightarrow SA | a$.
 Construct the SLR parse table for the grammar. Show the actions of the parser for the input string “aba”. **(C) (May/June 2014)**
13. What is an ambiguous grammar? Is the following grammar ambiguous? Prove
 $E \rightarrow E + E | E * E | (E) | id$. **(E) (May/June 2014)**
14. Generate SLR Parsing table for the following grammar.
 $S \rightarrow As | bAc | Bc | bBa$
 $A \rightarrow d$
 $B \rightarrow d$ **(C) (Apr/May 2015)**
15. Write down the algorithm to eliminate left recursion and left factoring and apply both to the following grammar. **(C) (Apr/May 2015)**
 $E \rightarrow E + T | E * | T$
 $T \rightarrow a | b | (E)$

16. Find the LALR for the given grammar and parse the sentence $(a+b) * c$. (C)
 $E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid id$. (Nov/Dec 2014)
17. What is an ambiguous grammar? Is the following grammar ambiguous? Prove
 $E \rightarrow E + E \mid E * E \mid (E) \mid id$. (AN)(May/June 2014)
18. Construct stack implementation of shift reduce parsing for the grammar
 $E \rightarrow E + S$
 $E \rightarrow E * E$
 $E \rightarrow (E)$
 $E \rightarrow id$ and the input string $id1 + id2 * id3$ (C)(MAY/JUNE 2016)
19. Explain LL(1) grammar for the sentence $S \rightarrow iEts \mid iEtSeS \mid aE \rightarrow b$ (U)(MAY/JUNE 2016)
20. Write an algorithm for Non recursive predictive parsing . (C)(MAY/JUNE 2016)
21. Explain Context Free grammars with examples (U)(MAY/JUNE 2016)
22. Distinguish between context free grammar and regular expressions. (A)(NOV/DEC 2015)
23. What are the conflicts during shift – reduce parsing? Explain (U)(NOV/DEC 2015)
24. Consider the grammar
 $E \rightarrow E + T \mid T$
 $T \rightarrow TF \mid F$
 $F \rightarrow F * a \mid b$
Construct the SLR parsing table for the above grammar. (C) (NOV/DEC 2016)(NOV/DEC 2021)
25. Construct parse tree for the input string $w = cad$ using topdown parser.
 $S \rightarrow cAd$
 $A \rightarrow ab \mid a$ (C)(NOV/DEC 2016)
26. Construct parsing table for the grammar and find moves made by predictive parser on input $id + id * id$ and find FIRST and FOLLOW.
 $E \rightarrow E + T$
 $E \rightarrow T$
 $T \rightarrow T * F$
 $T \rightarrow F$
 $F \rightarrow (E) \mid id$ (C) (NOV/DEC 2016)
27. Explain ambiguous grammar $G: E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid id$ for the sentence $id + id * id$. (U)(NOV/DEC 2016)
28. Construct SLR parsing Table for the following grammar:
 $G: E \rightarrow E + T \mid TT \rightarrow T * F \mid FF \rightarrow (E) \mid id$. (C) (NOV/DEC 2016)
29. Explain LR parsing algorithm with an Example. (R) (NOV/DEC 2017)
30. Explain the non recursive implementation of predictive parsers with the help of the grammar. (U) (NOV/DEC 2017)
- $E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid id$.
31. Consider the context-Free Grammar (CFG) depicted below where “begin”, “end” and “x” are all terminal symbols of the grammar and stat is considered the starting symbol for this

grammar. Productions are numbered in parenthesis and you can abbreviate “begin” to “b” and “end” to “e” respectively.

Block \rightarrow Block

Block \rightarrow begin Block end

Block \rightarrow Body

Body \rightarrow x

(i) Compute the set of LR(1) items for this grammar and draw the corresponding DFA. Do not forget to augment the grammar with the initial production $S \rightarrow \text{start}$ as the production (0).

(ii) Construct the corresponding LR parsing table. **(C) (APR/MAY 2018)**

32. Consider the following CFG grammar over the non-terminals $\{X, Y, Z\}$ and the terminals $\{a, c, d\}$ with the productions below and start symbol Z.

$X \rightarrow a$

$X \rightarrow Y$

$Z \rightarrow d$

$Z \rightarrow X Y Z$

$Y \rightarrow c$

$Y \rightarrow \epsilon$

Compute the FIRST and FOLLOW sets of every non-terminal and the set of non-terminals that are nullable. **(C) (APR/MAY 2018)**

33. Consider the following CFG grammar,

$S \rightarrow aABe$

$A \rightarrow Abc|b$

$B \rightarrow d$

Where a, b, c, d, e are terminals, 'S' (start symbol), A and B are non-terminals.

(a) Parse the sentence “abcde” using right most derivation

(b) Parse the sentence “abcde” using left-most derivations.

(c) Draw the parse tree. **(C) (APR/MAY 2018)**

34. Consider the following grammar

$E \rightarrow E+E|E*E|(E)|id.$

(i) Find the SLR parsing table for the given grammar.

(ii) Parse the sentence: $(a+b)*c$. **(C) (APR/MAY 2018)**

35. Check whether the following grammar is SLR (1) or not. Explain your answer with reasons.

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

36. Write grammar for the following languages the set of non-integers with no leading zeros. **(U)**

37. Write a context free grammar that generates all numbers; numbers can be integer or real. **(U)**

38. Write the algorithm for construction of LALR parsing table for a given grammar. Using the algorithm for construction of LALR parsing table construct the LALR parsing table for the following grammar. **(AP) (APR/MAY 2018)**

$S' \rightarrow S$
 $S \rightarrow aAd|bBd|aBe|bAe$
 $A \rightarrow c$
 $B \rightarrow c$

39. Show that the following grammar(**An**)(NOV/DEC 2018)

$S \rightarrow Aa/bAc/dc/bda$

$A \rightarrow a$ is LALR(1) but not SLR(1).

40. Show that the following grammar(**An**)(NOV/DEC 2018)

$S \rightarrow Aa/bAc/Bc/bBa$

$A \rightarrow d$

$B \rightarrow d$ is LR(1) but not LALR(1).

41. Construct LR(0) items for this following grammar and draw the transition Representing transition among CLR items

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

42. Show the whether the string “cdcd” is accepted by this grammar or not.

(i) What is SLR (1) parser. Describe the Steps for the SLR parser.

(ii) Give a rightmost derivation for (a, (a, a)) and show the handle of each right-sentential form.

43. Describe the LR parsing algorithm with an example(NOV/DEC 2021)

UNIT III INTERMEDIATE CODE GENERATION

Part- A

1. Give the syntax-directed definition for if-else statement. (U) (May/June 2011)

1. $S \rightarrow \text{if } E \text{ then } S1$

$E.\text{true} := \text{new_label}()$

$E.\text{false} := S.\text{next}$

$S1.\text{next} := S.\text{next}$

$S.\text{code} := E.\text{code} || \text{gen_code}(E.\text{true} ': ') || S1.\text{code}$

2. $S \rightarrow \text{if } E \text{ then } S1 \text{ else } S2$

$E.\text{true} := \text{new_label}()$

$E.\text{false} := \text{new_label}()$

$S1.\text{next} := S.\text{next}$

$S2.\text{next} := S.\text{next}$

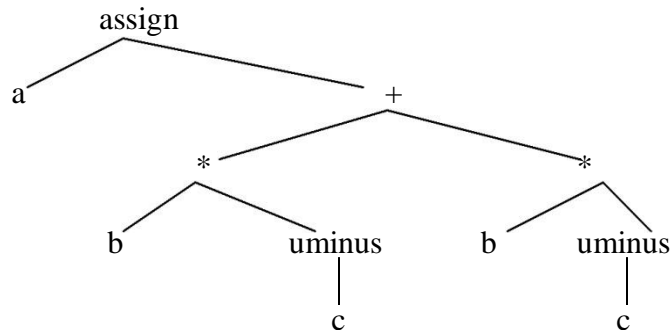
$S.\text{code} := E.\text{code} || \text{gen_code}(E.\text{true} ': ') || S1.\text{code} || \text{gen_code}(\text{'go to'}, S.\text{next}) ||$

$\text{gen_code}(E.\text{false} ': ') || S2.\text{code}$

2. What is a syntax tree? Draw the syntax tree for the assignment statement

$a := b * -c + b * -c.$ (U) (Nov/Dec 2012)

A syntax tree depicts the natural hierarchical structure of a source program.
Syntax tree:



3. Define procedure definition(R)

A procedure definition is a declaration that, in its simplest form, associates an identifier with a statement. The identifier is the procedure name, and the statement body. Some of the identifiers appearing in a procedure definition are special and are called formal parameters of the procedure. Arguments, known as actual parameters may be passed to a called procedure; they are substituted for the formal in the body.

4. Define activation trees(R)

A recursive procedure p need not call itself directly; p may call another procedure q, which may then call p through some sequence of procedure calls. We can use a tree called an activation tree, to depict the way control enters and leaves activation. In an activation tree

- Each node represents an activation of a procedure,
- The root represents the activation of the main program
- The node for a is the parent of the node for b if and only if control flows from activation a to b, and
- The node for a is to the left of the node for b if and only if the lifetime of a occurs before the lifetime of b.

5. Write notes on control stack(U)

A control stack is to keep track of live procedure activations. The idea is to push the node for activation onto the control stack as the activation begins and to pop the node when the activation ends.

6. Write the scope of a declaration(U)

A portion of the program to which a declaration applies is called the scope of that declaration. An occurrence of a name in a procedure is said to be local to if it is in the scope of a declaration within the procedure; otherwise, the occurrence is said to be nonlocal.

7. What do you mean by syntax directed translation scheme? (R)(APR/MAY 22)

For checking the semantics, each production of context-free grammar is related with a set of semantic rules or actions and each grammar symbol is related to a set of Attributes. Thus the grammar and the group of semantic actions combine to make syntax directed definitions.

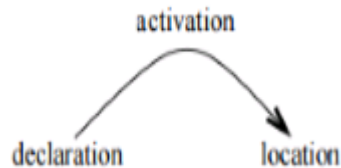
8. Write the translation scheme for (U)

```
while (i<10)
{ x := 0;
  i := i + 1;
}
S.begin = new_label() = L1
E.true = new_label() = L2
E.code = "if i<10 goto"
E.false = S.next = Lnext
S1.code = x = 0; i = i + 1
```

9. Draw the diagram of the general activation record and give the purpose of any two fields. (C)(April/May 2011)

- It is used to store the current record and the record is been stored in the stack.
- It contains return value. After the execution the value is been return.
- It can be called as return value. Parameter
- It specifies the number of parameters used in functions.

Activation Record:



An activation is an execution of a subprogram. In most languages, local variables are allocated when this execution begins (activation time).

The storage (for formals, local variables, function results etc.) needed for an activation is organized as an activation record (or frame).

10. Define Symbol Table. (R) (May/June 2014) (Nov/Dec 2016)

A Symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

11. Mention the rules for type checking. (R) (Apr/May 2017)

Express the rule for checking the type of a function (Nov/Dec 2021)

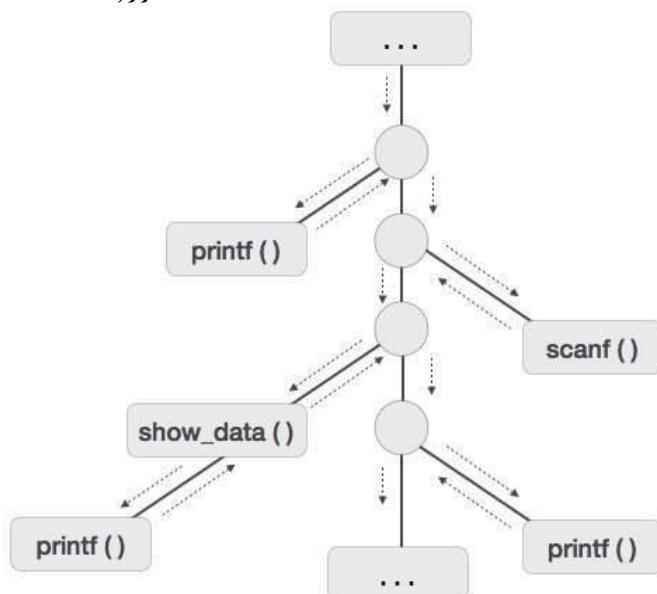
- Type Synthesis – Builds the type of an expression from the types of its subexpressions – Requires names to be declared before usage.
- Type inference – determines the type of a construct from the way it is used.

12. Write down syntax directed definition of a simple desk calculator.

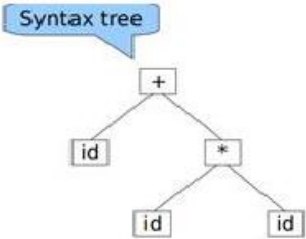
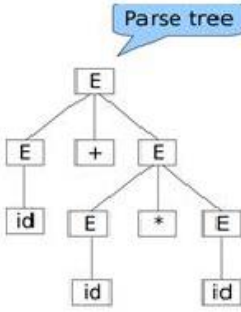
PRODUCTION	SEMANTIC RULE
$L \rightarrow En$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \text{digit}$	$F.val := \text{digit.lexval}$

13. Draw the activation tree for the following code:(C) (Apr/May 2018)

```
int main(){  
    printf(“enter your name”);  
    scanf(“%s”,username);  
    intshow_data(username);  
    printf(“Press any key to continue”);  
    ----  
    intshow_data(char *user)  
    {  
        printf(“Your name is %s”,username);  
        return 0;}}
```



14. Compare syntax tree and parse tree.(AN) (Nov/Dec 2017)

Syntax Tree	Parse Tree
interior nodes are “operators”, leaves are operands	interior nodes are non-terminals, leaves are terminals
when representing a program in a tree structure usually use a syntax tree	rarely constructed as a data structure
Represents the abstract syntax of a program (the semantics)	Represents the concrete syntax of a program
Contain only meaningful information.	Contain unusable information also.
Syntax tree examples Grammar: $E \rightarrow E * E \mid E + E \mid id$ Program: $a + b * c$ 	Parse tree examples Grammar: $E \rightarrow E * E \mid E + E \mid id$ Program: $a + b * c$ 

15. List three kinds of intermediate representation.(R)(Nov/Dec 2018)

There are varieties of forms to represent the intermediate code such as three address code, quadruple, triple, postfix.

16. When procedure call occurs, what are the steps taken? (Nov/Dec 2018)

When compiling a call to a procedure or function, each actual parameter is checked to see that it matches in kind and type with the corresponding formal parameter of the procedure's declaration.

17. What are the various ways of passing a parameter to a function? (Apr/May 2019)

Call-by-Value
 Call-by-Reference
 Copy-Restore
 Call-by-Name

18. Write the grammar for flow control statement while-do.(Apr/May 2019)

$S \rightarrow \text{while} (B) S_1$	<pre> begin = newlabel() B.true = newlabel() B.false = S.next S₁.next = begin S.code = label(begin) B.code label(B.true) S₁.code gen('goto' begin) </pre>
--	---

19. State the type expressions.(NOV/DEC 2021)

The type of a language construct is denoted by a type expression.

- A type expression can be: – A basic type (also called primitive types)
- a primitive data type such as integer, real, char, boolean, ...

Type name : a name can be used to denote a type expression

20. Convert the following statement into three address codes $x=a+(b*-c)+(d*-e)$

Represent the three address code by triples.(APR/MAY 2022)

t1=UMINUS(c)

t2=b*t1

t3=UMINUS(e)

t4=d*t3

t5=a+t2

t6=t5+t4

x=t6

PART-B

1. Give a syntax directed definition to differentiate an expression formed by applying the arithmetic operators * and to the variable x and constants.
Expression: $x * (3*x \mid x * x)$ **(AP) (Apr/May 2015)**
2. What is an activation record? Explain how it's relevant to the intermediate code generation phase with respect to procedure declarations. **(U) (Apr/May 2015)**
3. Discuss specification of a simple type checker **(A) (Apr/May 2017)**
4. Distinguish between quadruples and triples with examples. **(E)(Nov/Dec 2015)**
5. What are the rules for type checking? Give an example. **(R) (Nov/Dec 2015)**
6. State and explain the algorithm for unification. **(U) (Nov/Dec 2015)**
7. A Syntax-Directed Translation scheme that takes strings of a's, b's and c's as input and produces as output the number of substrings in the input string that correspond to the pattern $a(a|b)^*c+(a|b)^*b$. For example the translation of the input string "abbcabababc" is "3".
 - (1) Write a context – free grammar that generate all strings of a's, b's and c's.
 - (2) Give the semantic attributes for the grammar symbols.
 - (3) For each production of the grammar present a set of rules for evaluation of the semantic attributes. **(C) (Nov/Dec 2016)**
8. Illustrate type checking with necessary diagram. **(R) (Nov/Dec 2016)**
9. Discuss specification of a simple type checker for statement, expression and functions. **(R) (NOV/DEC 2017)**

10. Describe about the contents of activation record.(U) (APR/MAY 2018)(Nov/Dec 2021)
11. Create a parse tree for the following string:stringid+id-id.Check whether the string is ambiguous or not.(C) (APR/MAY 2018)
12. Construct a Syntax directed translation scheme that translates arithmetic expression from infix to postfix notation.Using semantic attributes for each of the grammar symbols and semantic rules,Evaluate the input: $3*4+5*2$.(C) (APR/MAY 2018) (Nov/Dec 2018)(Nov/Dec 2021)
13. Explain the design of predictive translator. (U)
14. Explain in detail a simple type checker. (U)
15. Explain the construction of syntax tree with an example. (U)
16. Construct a syntax directed definition for constructing a syntax tree for assignment statements (C)
 - $S \rightarrow is := E$
 - $E \rightarrow E1 + E2$
 - $E \rightarrow E1 * E2$
 - $E \rightarrow -E1$
 - $E \rightarrow (E1)$
 - $E \rightarrow id$
17. Differentiate call-by-value and Call-by-reference parameter passing mechanisms with suitable examples(U) (APR/MAY 2019)
18. Describe syntax-directed translation schemes with appropriate examples.(U) (APR/MAY 2019)
19. Explain how type conversion is performed with suitable examples.(U)(APR/MAY 2019)
20. Apply the S-attributed definition and constructs syntax trees for a simple expression grammar involving only the binary operators + and -. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminal have one synthesized attribute node, which represents a node of the syntax tree.
Production: $E \rightarrow E+T/T$, $T \rightarrow (E)/id/num$ (Nov/Dec 2018)
21. Suppose we have a production $A \rightarrow BCD$. Each of the four nonterminals has two attributes s, which is synthesized and I which is inherited. For each set of rules below, check whether the rules are consistent with (i) an S- attributed definition, (ii) an L-attributed definition (iii) any evaluation order at all. (Nov/Dec 2018)
 - (1) $A.s = B.i + C.i$
 - (2) $A.s = B.i + C.s$ and $D.i = A.i + B.s$
 - (3) $A.s = B.s + D.s$
 - (4) $A.s = D.i$
 - $B.i = A.s + C.s$
 - $C.i = B.s$
 - $D.i = B.i + C.i$
22. Create a parse trees for the following string : string id + id – id. Check whether the string is ambiguous or not. (7) (Nov/Dec 2021)

23. (i) Explain about various ways to pass a parameter in a function with example. (6)
(ii) Construct a Syntax-Directed Translation scheme that translates arithmetic expressions from infix into postfix notation. Using semantic attributes for each of the grammar symbols and semantic rules, Evaluate the input: $3*4+5*2$. (7)
(Nov/Dec 2021)
24. Elucidate the variants of Syntax tree with suitable examples. (Nov/Dec 2021)
25. (i) Write an algorithm for unification with its operation. (Nov/Dec 2021)
26. (ii) Discuss in detail about Translation of array reference. (Nov/Dec 2021)
27. Write the syntax directed translation for the following code
E \rightarrow E1 or E2
E \rightarrow E1 and E2
E \rightarrow not E1
E \rightarrow (E)
E \rightarrow id1 relop id2
E \rightarrow true
E \rightarrow false (Apr/May 2022)
28. Write the syntax directed translation for the following code:
While a < b
do
If c < d
Then
X = y + z
Then
X = y - z (Apr/May 2022)

UNIT IV RUN-TIME ENVIRONMENT AND CODE GENERATION

1. What is the use of run time storage? (U)

The run time storage might be subdivided to hold

- a) The generated target code
- b) Data objects, and
- c) A counterpart of the control stack to keep track of procedure activation.

2. What is an activation record? (R)

Information needed by a single execution of a procedure is managed using a contiguous block of storage called an activation record or frame, consisting of the collection of fields such as

- a) Return value
- b) Actual parameter
- c) Optional control link
- d) Optional access link
- e) Saved machine status

- f) Local data
- g) Temporaries

3. What are the storage allocation strategies? (R)(NOV/DEC 2017)

Name different storage allocation strategies used in runtime environment.(R)(APR/MAY 2018)(NOV/DEC 2021)

List the different storage allocation strategies.(NOV/DEC 2021)ACTIVATION

- a) Static allocation lays out storage for all data objects at compile time.
- b) Stack allocation manages the run time storage as a stack.
- c) Heap allocation allocates and deallocates storage as needed at run time from a data area known as heap.

4. What is static allocation? (R)

In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package. Since the bindings do not change at run time, every time a procedure is activated, its names are bound to the same storage location.

5. What is stack allocation? (R) (May/June 2008)

Stack allocation is defined as process in which manages the run time as a Stack. It is based on the idea of a control stack; storage is organized as a stack, and activation records are pushed and popped as activations begin and end.

6. What are the limitations of static allocation? (U)

- a) The size of a data object and constraints on its position in memory must be known at compile time.
- b) Recursive procedure is restricted.
- c) Data structures cannot be created dynamically.

7. What is dangling references? (R) (May/June 2016)

What does dangling references mean? (Apr/May 2022)

Whenever storage can be deallocated, the problem of dangling references arises. A dangling reference occurs when there is a reference to storage that has been deallocated.

8. What is heap allocation? (R)

Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects. Pieces may be deallocated in any order, so over time the heap will consist of alternate areas that are free and in use.

9. What is dangling references? (R) (May/June 2016)

Whenever storage can be deallocated, the problem of dangling references arises. A dangling reference occurs when there is a reference to storage that has been deallocated.

10. What is heap allocation? (R)

Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects. Pieces may be deallocated in any order, so over time the heap will consist of alternate areas that are free and in use.

11. Define Garbage. (R)

Dynamically allocated storage can become unreachable. Storage that a program allocates but cannot refer to is called garbage. Lisp performs garbage collection that reclaims inaccessible storage.

12. List the dynamic storage allocation techniques? (R)(Nov/Dec 2016)

- a) Explicit allocation of Fixed-sized blocks.
- b) Explicit allocation of Variable-sized blocks—one method is first-fit method, in this when a block of size s is allocated; we search for the first free block that is of size $f \geq s$.
- c) Implicit Deallocation—it requires cooperation between the user program and the run-time package.

13. What are the limitations of static allocation? (U) (May/June 2009)

It lays out storage for all data objects at compile time.
Names are bound to storage as a program is compiled, so there is no need for a run time support package.

14. What do you mean by binding of names(U)(Apr/May 2017)

When an environment associates storage location s with a name x , we say that x is bound to s ; the association itself is referred to as a binding of x . A binding is the dynamic counterpart of a declaring.

15. What is meant by call-by-reference? (U)

When parameters are passed by reference, the caller passes to the called procedure a pointer to the storage address of each actual parameter.

- a) If an actual parameter is a name or an expression having l-value, then that l-value itself is passed.
- b) However, if the actual parameter is an expression, then the expression is evaluated in a new location, and address of that location is passed.

16. What is meant by copy-restore? (R)

A hybrid between call-by-value and call by reference is copy-restore linkage.

- 1. Before control flows to the called procedure, the actual parameters are evaluated.
- 2. When control returns, the current r-values of the formal parameters are copied back into the l-value s of the actual, using the l-values computed before the call.

17. Write notes on call-by-name. (U)

Call-by-name is traditionally defined by the copy-rule of Algol, which is

- a) The procedure is treated as if it were a macro; that is, its body is substituted for the call in the caller, with the actual parameters literally substituted for the formals. Such a literal substitution is called macro-expansion or in-line expansion.
- b) The local named of the called procedure are kept distinct from the names of the calling procedure.
- c) The actual parameters are surrounded by parenthesis if necessary to preserve their integrity.

18. State the problems in code generation. (R)(Nov/Dec 2018)

Input to code generator
Target program

Memory Management
Instruction selection
Register allocation issues
Evaluation order
Approaches to code generation issues

19. Define address descriptor.(U) (Apr/May 2019)

Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

20. Write the object code sequence for $t:=a+b$ produced by a typical code generator.(U)(Apr/May 2019)

MOV a,R0
ADD b, R0
MOV R0,t

PART B

1. Discuss in detail about the run time storage arrangement. (U)(NOV/DEC 2017)
2. Describe about the stack allocation in memory management. (R)
3. What are the different storage allocation strategies? (R) (May/June 2013)(Apr/May 2017)(Nov/Dec 2018)(APR/MAY 2019)
4. Mention in detail any 4 issues in storage organization. (R) (Apr/May 2015)
5. Describe the various storage allocation strategies. (R)(Nov/Dec 2008) (May/June 2009)
6. Describe in detail the source language issues. (R) (Nov/Dec 2007)
7. Explain in detail access to nonlocal names. (U)
8. Elaborate storage organization. (U) (May/June 2013)
9. What are the storage allocation strategies? Explain them with example. (U) (Nov/Dec 2015)
10. Distinguish between static and dynamic storage allocations. (E) (Nov/Dec 2015)
11. Explain the one pass code generation using back patching with Examples.(U) (Nov/Dec 2015)
12. Explain the following with respect to code generation phase.(R)(Nov/Dec 2016)
 - (i) Input to code generator
 - (ii) Target program
 - (iii)Memory management
 - (iv) Instruction selection
 - (v)Register allocation
 - (vi) Evaluation order.
13. Write in detail about storage allocation in FORTRAN. (R)
14. Explain various issues in the design of code generator. (U) (MAY/JUNE 2016)(Apr/May 2017)
15. Write the algorithm for a simple code generator. (C) (MAY/JUNE 2016)(Nov/Dec 2021)

16. Write the Code Generation Algorithm using Dynamic Programming and generate code for the statement $x = a/(b-c) - s*(e+1)$ [assume all instructions to be unit cost](C)(Apr/May 2015)
17. Discuss the issues in code generation with example.(R)(Nov/Dec 2017)(Nov/Dec 2021)
18. Translate the following assignment statement into three address code.
 $D := (a-b)*(a-c) + (a-c)$
 Apply code generation algorithm, generate a code sequence for the three address statement.(C)(APR/MAY 2018)
19. Summarize the issues arise during the design of code generator.(R)(Apr/May 2018)(Nov/Dec 2018)
20. Write detailed notes on parameter passing.(R)(Apr/May 2017)(Nov/Dec 2018)
21. Explain the algorithm that generates code for a single basic block with suitable examples.
 (U)(APR/MAY 2019)
22. (a) Elaborate the issues in design of a code Generator(APR/MAY 2022)
23. Construct the basic block and flow graph for the following piece of code
 for i from 1 to 10 do
 for j from 1 to 10 do
 $a[i,j] = 0.0$
 for i from 1 to 10 do
 $a[i,i] = 1.0$ (APR/MAY 2022)
24. Discuss in detail about stack allocation space of memory and the usage of stack in the memory allocation. (NOV/DEC 2021)
25. Elaborate the various issues in code generation with examples. (NOV/DEC 2021)

UNIT V CODE OPTIMIZATION

Part- A

1. Define peephole optimization. (R)

Peephole optimization is a simple and effective technique for locally improving the target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions, and replacing these instructions by shorter or faster sequence

2. List the characteristics of peephole optimization(R)(May/June 2007)

(Nov/Dec2016)Point out the characteristics of peephole optimization.(Nov/Dec 2021)

Identify and write down the optimizations that could be performed on a peephole.(APRIL/MAY 2022)

- Redundant instruction elimination
- Flow of control optimization
- Algebraic simplification
- Use of machine idioms
- Reduction in strength

3. How do you calculate the cost of an instruction? (E) (Nov/Dec 2010)

The cost of an instruction can be computed as one plus the cost associated with the source and destination addressing modes given by added cost.

```
MOV R0,R1 1
MOV R1,M 2
SUB 5(R0),*10(R1) 3
```

4. What is a basic block? (R) (Nov/Dec 2010) (Apr/May 2017)

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching.

```
Eg. t1:=a*5
t2:=t1+7
t3:=t2-5
t4:=t1+t3
t5:=t2+b
```

5. How would you represent the following equation using DAG? (E) (May/June 2013)

$a := b * -c + b * -c$

6. Mention the issues to be considered while applying the techniques for code optimization. (A)

- The semantic equivalence of the source program must not be changed.
- The improvement over the program efficiency must be achieved without changing the algorithm of the program.

7. What are the basic goals of code movement? (R)

To reduce the size of the code, i.e. to obtain the space complexity.

To reduce the frequency of execution of code i.e. to obtain the time complexity.

8. What do you mean by machine dependent and machine independent optimization? (U)

The machine dependent optimization is based on the characteristics of the target machine for the instruction set used and addressing modes used for the instructions to produce the efficient target code.

The machine independent optimization is based on the characteristics of the programming languages for the appropriate programming structure and usage of efficient arithmetic properties in order to reduce the execution time.

9. What are the different data flow properties? (R)

- Available expressions
- Reaching definitions
- Live variables
- Busy variables

10. What is code motion? (R) (May/June 2007) (May/June 2008)

Code motion is an optimization technique in which amount of code in a loop is decreased. This transformation is applicable to the expression that yields the same result independent of the number of times the loop is executed. Such an expression is placed before the loop.

11. What are the properties of optimizing compiler?(R)(MAY/JUNE 2016)(NOV/DEC 2021)

The source code should be such that it should produce a minimum amount of target code. There should not be any unreachable code.

Dead code should be completely removed from the source language.

The optimizing compilers should apply following code improving transformations on the source language.

- i) Common sub expression elimination
- ii) Dead code elimination
- iii) Code movement
- iv) Strength reduction

12. Explain the principle sources of optimization. (U) (May/June 2012)

Code optimization techniques are generally applied after syntax analysis, usually both before and during code generation. The techniques consist of detecting patterns in the program and replacing these patterns by equivalent and more efficient constructs.

13. What are the 3 areas of code optimization? (R) (May/June 2012)

- Local optimization
- Loop optimization
- Data flow analysis

14. Define local optimization. (R)

The optimization performed within a block of code is called a local optimization.

15. Define constant folding. (R) (May/June 2013) (APRIL/MAY 2022)

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

16. What do you mean by inner loops? (U)

The most heavily traveled parts of a program, the inner loops, are an obvious target for optimization. Typical loop optimizations are the removal of loop invariant computations

and the elimination of induction variables.

17. What is code motion? (R) (April/May 2004, May/June 2007, April/May-2008)

Code motion is an important modification that decreases the amount of code in a loop.

18. Define Local transformation & Global Transformation. (R)

A transformation of a program is called Local, if it can be performed by looking only at the statements in a basic block otherwise it is called global.

19. Give examples for function preserving transformations. (U) (May/June 2010)

- Common subexpression elimination
- Copy propagation
- Dead – code elimination
- Constant folding

20. What is meant by Common Subexpressions? (R) (Nov/Dec 2018)

An occurrence of an expression E is called a common subexpression, if E was previously computed, and the values of variables in E have not changed since the previous computation.

21. What is meant by Dead Code? (R)

A variable is live at a point in a program if its value can be used subsequently otherwise, it is dead at that point. The statement that computes values that never get used is known Dead code or useless code.

22. Mention various techniques used for loop optimization? (R)(APR/MAY 2018)

Code motion
Induction variable elimination
Reduction in strength

23. What is meant by Reduction in strength? (U)

Reduction in strength is the one which replaces an expensive operation by a cheaper one such as a multiplication by an addition.

24. What is meant by loop invariant computation? (U)

An expression that yields the same result independent of the number of times the loop is executed is known as loop invariant computation.

25. Define data flow equations. (R) (Nov/Dec 2010)

A typical equation has the form

$$\text{Out}[S] = \text{gen}[S] \cup (\text{In}[S] - \text{kill}[S])$$

and can be read as, “ the information at the end of a statement is either generated within the state, or enter at the beginning and is not killed as control flows through the statement”. Such equations are called data flow equations.

26. What is a block? Give its syntax. (R)

A block is a statement containing its own data declaration.

Syntax:

```
{  
Declaration statements  
}
```

27. Write three address code sequence for the assignment statement

d:=(a-b)+(a-c)+(a-c).(C)(MayY/June 2016) (NOV/DEC 2021)

28. Mention the criteria for code-improving transformations. (R) (Nov/Dec 2008)

A transformation must preserve meaning of a program (correctness)

A transformation must improve (e.g., speed-up) programs by a measurable amount on average transformation must worth the effort Indicate the places for potential improvements can be made by the user and the compiler.

29. Mention the function preserving, code improving transformations.(R)(Nov/Dec2011)

Simply stated, the best program transformations are those that yield the most benefit for the least effort.

First,the transformation must preserve the meaning of programs. That is,the optimization must not change the output produced by a program for a given input,or cause an error.

Second,a transformation must,on the average,speed up programs by a measurable amount.

Third,the transformation must be worth the effort.Some transformations can only be applied after detailed,often time-consuming analysis of the source program,so there is little point in applying them to programs that will be run only a few times.

30. What is code motion? Give an example. (R) (Nov/Dec 2010)

Code motion is an optimization technique in which amount of code in a loop is decreased. This transformation is applicable to the expression that yields the same result independent of the number of times the loop is executed. Such an expression is placed before the loop.

31. What is constant folding? (R)(May/June 2013)

Constant folding is the process of replacing expressions by their value if the value can be computed at complex time.

32. Would you represent the dummy blocks with no statements indicated in the global data flow analysis? (A) (May/June 2014)

- Region consisting of a statement S:Control can flow to only one block outside the region
- Loop is a special case of a region that is strongly connected and includes all its back edges.
- Dummy blocks with no statements are used as technical convenience (indicated as open circles)
-

33. Define live variable. (R)

When do you call a variable to syntactically live at appoint?(Apr/May 2022)

A variable is live at a point in a program if its value can be used subsequently.

34. What do you mean by copy propagation(U)(Apr/May 2017)

Copy propagation is the process of replacing the occurrences of targets of direct assignments with their values.

The assignment statement of the form $f := g$ is called copy statements.

The common subexpression $c := d + e$ is eliminated

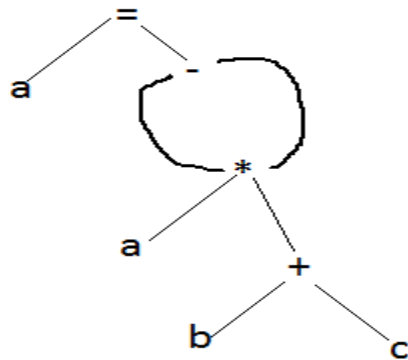
The algorithm uses a new variable t to hold the value of $d + e$.

Since control may reach $c := d + e$ either after the assignment of a or after the assignment of b it would be incorrect to replace $c := d + e$ by either $c := a$ or $c := b$.

35. Identify the constructs for optimization in basic block. (U) (Nov/Dec 2016)

Structure-preserving transformations

- Common sub expression elimination
- Dead-code elimination
- Algebraic transformations
- Reduction in strength.

36. Draw the DAG for the statement $a := (a * b + c) - (a * b + c)$ (C) (NOV/DEC 2017)**37. What are the properties of optimizing compiler? (R) (NOV/DEC 2017)**

1. A transformation must preserve the meaning of programs.
2. A transformation must speed up programs by a measurable amount.
3. A transformation must be worth the effort.

38. List out the primary structure preserving transformations on basic block.(R) (APR/MAY 2018)

What are the structure preserving transformations on basic blocks? (NOV/DEC 2021)

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

39. What do you mean by DAG? (U)

It is Directed Acyclic Graph. In this common sub expressions are eliminated. So it is a compact way of representation.

PART – B

1. Explain the principle sources of optimization in detail. (U)
(May/June 2013, Nov/Dec 2014) (Nov/Dec 2021)
2. Discuss about the following:
i). Copy Propagation ii) Dead-code Elimination and iii) Code motion (R) (Nov/Dec 2010)
3. Write about Data flow analysis of structured programs. (C)(Nov/Dec 2007)
4. Explain optimization of basic blocks. (U)(Apr/May 2017)(Nov/Dec 2007,2018)
5. Discuss in detail the process of optimization of basic block. Give an example. (U)
(May/June 2014, Nov/Dec 214)
6. What is data flow analysis? Explain data flow abstraction with examples. (U)
(May/June 2014)
7. Explain loop optimization and apply it to the code given below. (A) (Apr/May 2015)
 - (i) i:=0
 - (ii) a:=n_3
 - (iii) IF i<a THEN loop ELSE end
 - (i) LABELL Lop
 - (ii) b:=i_4
 - (iii) c:=p+b
 - (iv) d:=M[c]
 - (v) e:=d_2
 - (vi) f:=i_4
 - (vii) g:=p+f
 - (viii) M[g]:=e
 - (ix) I:=i+1
 - (x) a:=n_3
 - (xi) IF i<a THEN loop ELSE endLABELL end
8. What are the optimization techniques applied to procedure calls? Explain with example.
(U) (Apr/May 2015)
9. Construct DAG and optimal target code for the expression(C)(Apr/May 2015)
 $x=((a+b)/(b-c))-(a+b)*(b-c)+/.$
10. Explain peephole optimization and various code improving transformations. (U)
(Nov/Dec 2014)
11. What are the advantages of DAG representation? Give example. (R) (Apr/May 2015)
12. Explain principal sources of optimization with examples. (U)(MAY/JUNE 2016)(NOV/DEC 2017)
13. Explain the procedure to find the induction variable in loops and optimize their computation. What is the function of strength reduction? (U) (NOV/DEC 2015)
14. What is peephole optimization? State and explain the characteristic of peephole optimization. (U)(NOV/DEC 2015)
18. What is data flow abstraction? Explain it with a program illustrating the data Flow abstraction. (U)(NOV/DEC 2015)
19. What is live-variable analysis? Explain it with example.(R)(NOV/DEC 2015)
20. Construct DAG for the following Basic Block.
 1. t1:=4*i
 2. t2:=a[t1]

3. $t3:=4*i$
4. $t4:=b[t3]$
5. $t5:=t2*t4$
6. $t6:=prod+t5$
7. $Prod:=t6$
8. $t7:=i+1$
9. $i:=t7$

10. If $i \leq 20$ goto (1)(C)(APR/MAY 2017)

21. Write an algorithm for constructing natural loop of a back edge.(C)(NOV/DEC 2016)
22. Explain any four issues that crop up when designing a code generator.(R) (NOV/DEC 2016)
23. Explain global data flow analysis with necessary equations.(R) (NOV/DEC 2016)
24. Determine the basic block of instructions, Control Flow Graph (CFG) and the CFG dominator tree for following code.

```

01 a=1
02 b=0
03 L0: a=a+1
04     B=p+1
05     If(a>b) goto L3
06 L1: a=3
07     If(b>a) goto L2
08     B=b+1
09     Goto L1
10 L2: a=b
11     b=p+q
12     If(a>b) goto L0
13 L3: t1=p*q
14     t2=t1+b
15     return t2

```

(C)(APR/MAY 2018)

25. Construct a code sequence and DAG for the following syntax directed translation of the expression: $(a+b)-(e-(c+d))$ (C)(APR/MAY 2018)
 26. Draw the symbol tables for each of the procedures in the following PASCAL code (including main) and show their nesting relationship by linking them via a pointer reference in the structure (or record) used to implement them in memory. Include the entries or fields for the local variables, arguments and any other information you find relevant for the purpose of code generation, such as its types and location at run time.
- ```

01 procedure main
02 integer a,b,c;
03 procedure f1(a,b)
04 integer a,b;
05 call f2(b,a);
06 end;

```

```

07 procedure f2(y,z);
08 integer y,z;
09: procedure f3(m,n);
10: integer m,n;
11: end;
12: procedure f34(m,n);
13: integer m,n;
14: end;
15: call f3(c,z);
16: call f4(c,z);
17: end;
18: ...
19: call f1(a,b);
20: end;

```

**(C) (APR/MAY 2018)**

27. Construct the DAG for the following basic block. **(C) (APR/MAY 2019)**

```

x=a[i]
a[j]=y
z=a[i]

```

28. Write and explain the algorithm for construction of basic blocks. **(U) (APR/MAY 2019)**

29. A simple matrix-multiplication program is given below: **(C) (APR/MAY 2019)**

```

for (i=0;i<n;i++)
for (j=0;j<n;j++)
c[i][j]=0.0;
for (i=0;i<n;i++)
for (j=0;j<n;j++)
for (k=0;k<n;k++)
c[i][j]=c[i][j]+a[i][k]*b[k][j];

```

- (i) Translate the program into three-address statements. Assume the matrix entries are numbers that require 8 bytes, and that matrices are stored in row-major order.

- (ii) Construct the flow graph for the code from 1.

- (iii) Identify the loops in the flow graph from 2. **(Nov/Dec 2021)**

30. Describe the parameter passing technique with an Example **(APR/MAY 2022)**

31. Explain the storage allocation techniques with an example **(APR/MAY 2022)**

32. Consider the following basic block, in which all variables are integers and \*\* denotes exponentiation

```

a:=x**s
b:=3
c:=x
d:=c*c
e:=b*s
f:=a+d

```

$$g := e * f$$

Apply the following optimization techniques to this basic block in order. Compute the result of each transformation.

- (i) Algebraic Simplification
- (ii) Copy propagation
- (iii) Constant Folding
- (iv) Dead code Elimination
- (v) Common sub-expression elimination

33. Formulate the steps for efficient Data Flow algorithm. **(NOV/DEC 2021)**

34. Describe the representation of array using DAG with example. **(NOV/DEC 2021)**

35. Summarize in detail about the global dataflow analysis with example. **(NOV/DEC 2021)**

36. Apply code generation algorithm to generate a code sequence for the three address statement for the following assignment statement.

$$D := (a - b) * (a - c) + (a - c)$$

**(NOV/DEC 2021)**