COMPILER DESIGN

Introduction To Compiler	2
Lexical Analysis	14
Parsing And Context Free Grammar	24
Operator Precedence Parsing	36
Top-Down Parsing	41
Bottom-Up Parsing and LR Parser Generation Theory	53
Syntax Directed Translation	72
Type Checking	87
Runtime Environment	88
Intermediate Code Generation	93
Code Optimization	106

NOTE:

MAKAUT course structure and syllabus of 5th semester has been changed from 2020. Previously **COMPILER DESIGN** was in 7th Semester. This subject has been shifted in 5th semester in present curriculum. Subject organization has been changed slightly. Taking special care of this matter we are providing chapterwise relevant MAKAUT university solutions, so that students can get an idea about university questions patterns.

INTRODUCTION TO COMPILER

Multiple Choice Type Questions

1. Which one of the following error will not be detected by the compiler? [WBUT 2010, 2012, 2016]
   a) Lexical error
   b) Syntactic error
   c) Semantic error
   d) Logical error

Answer: (d)

2. Which is used to keep track of currently active activations? [WBUT 2012]
   a) control stack
   b) activation
   c) execution
   d) symbol

Answer: (a)

3. A compiler that runs on one machine and produces code for a different machine is called [WBUT 2013]
   a) Cross compilation
   b) One pass compilation
   c) 2 pass compilation
   d) none of these

Answer: (a)

4. Which of the following translation program converts assembly language program to object program? [WBUT 2014]
   a) assembler
   b) compiler
   c) macro-processor
   d) linker

Answer: (a)

5. Role of preprocessor is to [WBUT 2015]
   a) produce output data
   b) produce output to compilers
   c) produce input to compilers
   d) none of these

Answer: (c)

6. Consider the program statement $b = 2$ where b is a Boolean variable. Which stage of compilation can detect this error? [WBUT 2015]
   a) Lexical analysis
   b) Syntax analysis
   c) Semantic analysis
   d) Code generation

Answer: (c)

7. Cross-compiler is a compiler [WBUT 2017]
   a) which is written in a language that is different from the source language
   b) that generates object code for host machine
   c) which is written in a language that is same as the source language
   d) that runs on one machine but produces object code for another machine

Answer: (d)

8. An ideal compiler should **[WBUT 2017]**

a) be smaller in size
b) take less time for compilation
c) be written in a high level language
d) produce object code that is smaller in size and executes faster

Answer: (d)

9. Compiler can check ..... Error. **[WBUT 2019]**

a) logical b) syntax c) content d) both (a) and (b)

Answer: (b)

Short Answer Type Questions

1. With the help of a block diagram, show each phase including symbol table and error handler of a compiler. **[WBUT 2007]**

OR,

Using a block diagram indicate the phases of compilation explaining their activities. **[WBUT 2014]**

OR,

Describe analysis phase of a Compiler with a block diagram. **[WBUT 2018]**

OR,

Describe the errors encountered in different phases of compiler. **[WBUT 2019]**

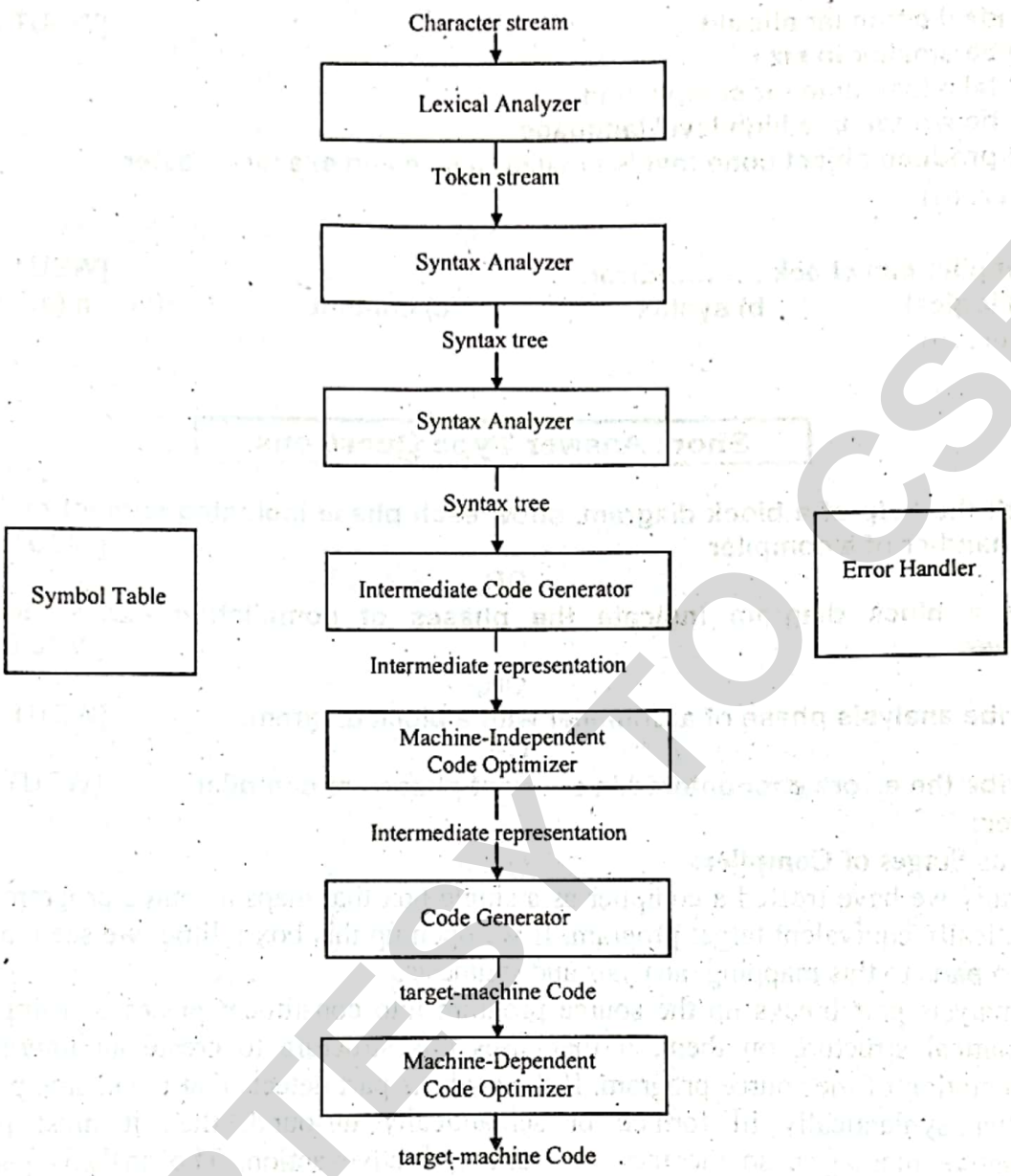
Answer:

Various Stages of Compiler:

Generally we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

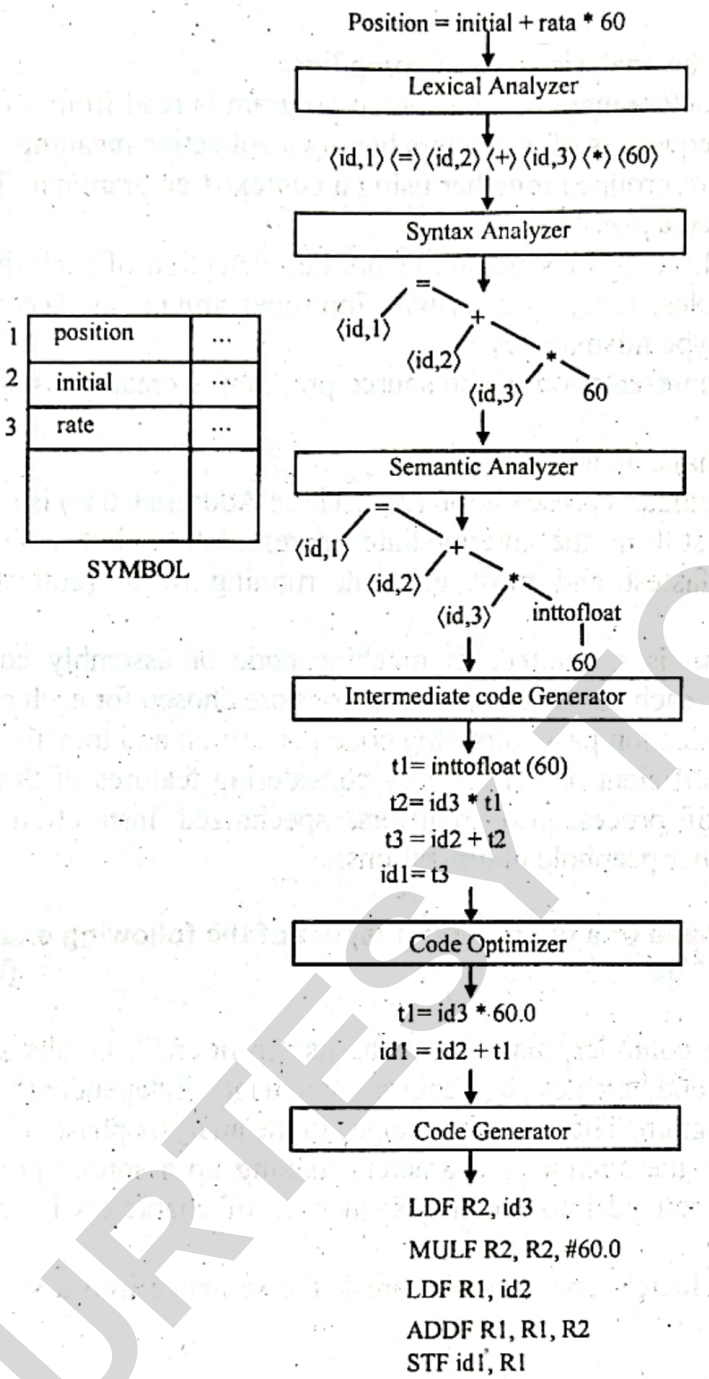
If we examine the compilation process in more details, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases in show in figure below. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the entire source program, is used by all phases of the compiler.



Phases of a compiler

Some compilers have a machine-independents optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an optimization intermediate representation. Since optimization is optional, one or the two optimization phases shown in above figure may be missing.

All phases are described using an example:



Translation of an assignment statement

2. What are the analysis phase and synthesis phase of an assemble? [WBUT 2010]

Answer:

There are two main stages in the compiling process --- analysis and synthesis.

The analysis stage breaks up the source program into pieces, and creates a generic (language-independent) intermediate representation of the program. Then, the synthesis stage constructs the desired target program from the intermediate representation. Usually,

the analysis stage of a compiler is called its front-end and the synthesis stage is its back-end.

There are four phases in the analysis stage of compiling:

Here, the stream of characters making up a source program is read from left to right and grouped into *tokens* --- sequences of characters having a collective meaning.

Here, the tokens found are grouped together using a context-free grammar. The output of this parsing phase is called a *parse tree*.

Here, the parse tree is checked for semantic errors i.e., detection of such things like the use of undeclared variables, function calls with improper arguments, access violations, incompatible operands, type mismatches, etc.

Here, the intermediate representation of the source program is created, usually in a form like *three-address code*.

The synthesis stage can have up to three phases:

Here, code in the intermediate representation (e.g., Three Address Code) is converted into a streamlined version, still in the intermediate representation, but with attempts to produce the smallest, fastest and most efficient running result (collectively called *optimization*).

Here, the target program is generated, as machine code or assembly code. Memory locations are selected for each variable and instructions are chosen for each operation.

This is an optional optimization pass following code generation and transforms the object code into tighter, more efficient object code by considering features of the hardware to make efficient usage of processor(s), registers, specialized instructions, pipelining, branch prediction and other peephole optimizations.

3. Describe analysis phase of a compiler in respect of the following example.
position = initial + rate * 60 **[WBUT 2011]**

Answer:

The analysis phase of a compiler, also known as its "front-end", breaks up the source program into pieces, and creates a generic (language independent) intermediate representation of the program. There are four stages in the analysis phase of compiling:

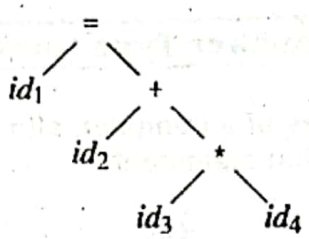
Lexical Analysis: Here, the stream of characters making up a source program is read from left to right and grouped into tokens—sequences of characters have a collective meaning.

For the given example, lexical analysis will break the sentence into a stream of seven tokens:

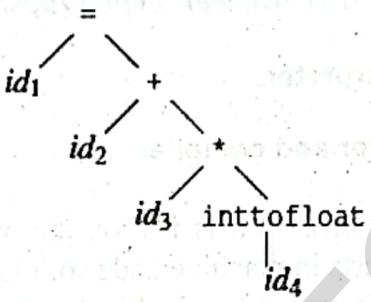
id1 = id2 + id3 _ id4

id1, id2 and id3 will be 'bound' to Symbol Table entries corresponding to variables position, initial, and rate, respectively, while id4 will be bound to the constant 60.

Syntax Analysis or Parsing: Here, the tokens found are grouped together using a context-free grammar. The output of this parsing phase is called a parse tree. The parse tree for the given example will be:



Semantic Analysis: Here, the parse tree is checked for semantic errors i.e., detection of such things like the use of undeclared variables, function calls with improper arguments, access violations, incompatible operands, type mismatches, etc. Assuming that in our code the variables are of float type, the semantic analysis will insert a int-to-float conversion of the constant 60 to give:



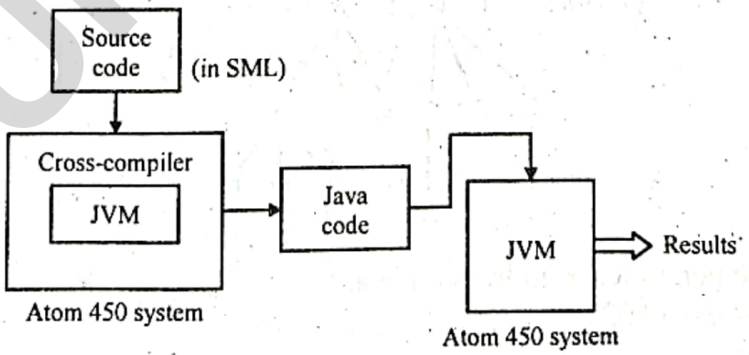
Intermediate Code Generation: Here, the intermediate representation of the source program is created, usually in a form like three-address code. The set of three-address codes for the example will be

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

4. What is a cross-compiler? Create a cross-compiler for SML (Sensor Mark-up Language) using Java compiler, written in ATOM-450, producing code in ATOM-450 and a SML language producing code for XML written in Java. [WBUT 2012]

Answer:

A cross-compiler is a piece of software which runs on a particular environment (processor + OS) and generates code that can run on another environment. For example a 'C' cross-compiler running on X86-Windows generating code for ARM-Linux.



Long Answer Type Questions

1. a) Explain the different phases of a compiler, showing the output of each phase, using the example of the following statement: [WBUT 2009]
Position: = initial + rate * 60

OR,

How the following statement is translated via the different phases of compilation?
Position : = initial + rate *70. [WBUT 2010]

OR,

Explain different stages of compiler with a suitable example. [WBUT 2013]

Answer:

Refer to Question No. 3 of Short Answer Type Questions.

b) Compare compiler and interpreter. [WBUT 2009]

OR,

Distinguish between interpreter and compiler. [WBUT 2012]

Answer:

Compiler scans whole code at once so it is fast on the other hand interpreter scans the byte code line by line and converts in machine code so it is slow.

Compiler creates executable file that runs directly on the CPU but interpreter does not create direct executable file.

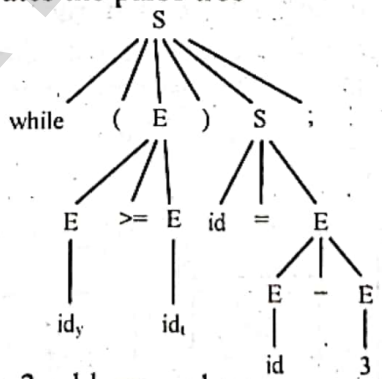
Compiler uses more memory - all the execution code needs to be loaded into memory.

Interpreter uses less memory, source code only has to be present one line at a time in memory.

2. a) Apply all the phases of compiler and show the corresponding output in every phase for the following code of the source program:
while (y ≥ t) y = y – 3; [WBUT 2012]

Answer:

Lexical analysis generates the tokens
'while', 'C', 'id_y', '>=', 'id_t', '=', 'id_y', '–', '3' and ';'
The syntax analysis stage generates the parse tree



The intermediate code generates 3-address code as:

100 : if (y>=t) goto 102
101 : goto 105
102 : T=y–3

103 : Y=T
104 : goto 100

The code optimizer optimized this code to

100 : if (y<t) go to 103
101 : y=y–3
102 : goto 100

The code generator can translate this into machine language statements of the target machine.

b) What do you mean by passes of compiler? Explain advantages and disadvantages of one-pass and two-pass over each other. [WBUT 2012]

Answer:

One-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. This is in contrast to a **multi-pass compiler** which converts the program into one or more intermediate representations steps in between source code and machine code, and which reprocesses the entire compilation unit in each sequential pass.

Advantages

One-pass compilers are smaller than Two-pass compiler and faster than multi-pass compilers.

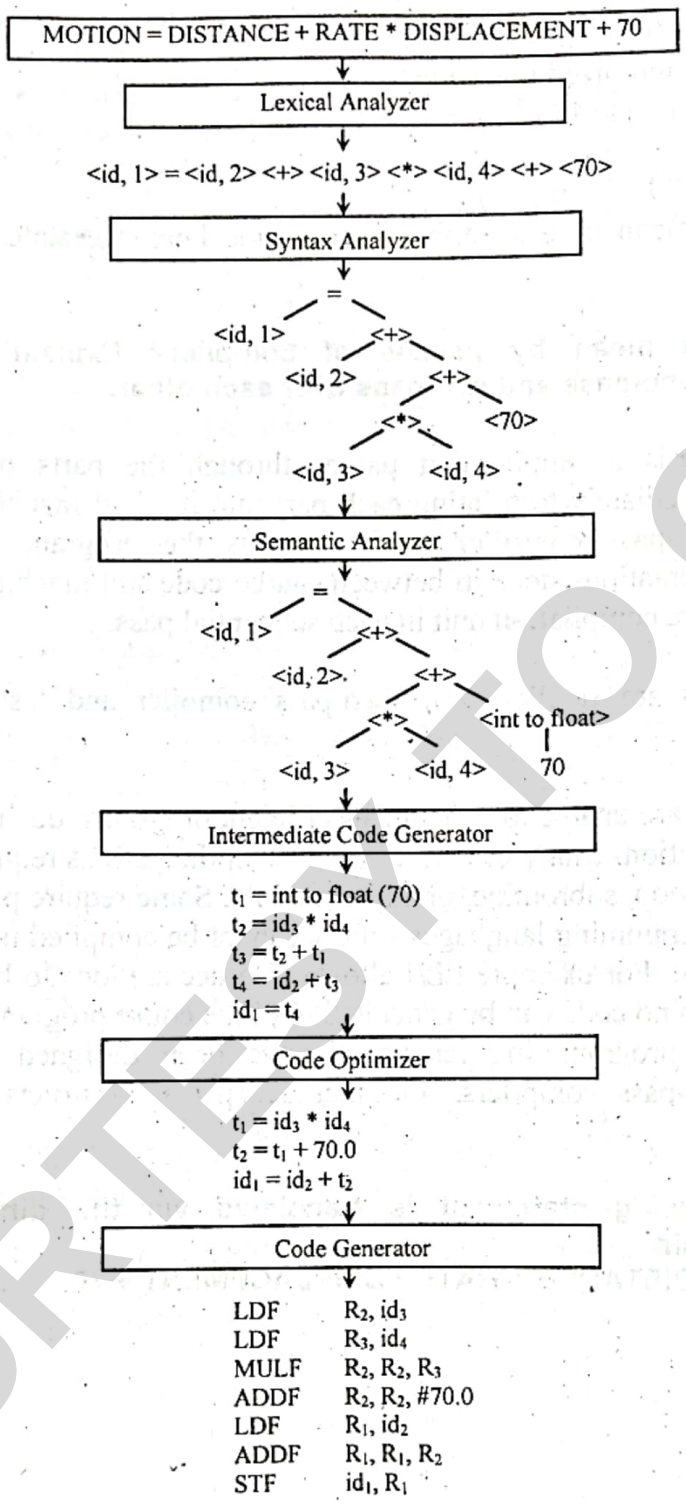
Disadvantages

One-pass compilers are unable to generate as efficient programs, due to the limited scope of available information. Many effective compiler optimizations require multiple passes over a basic block, loop, subroutine, or entire module. Some require passes over an entire program. Some programming languages simply cannot be compiled in a single pass, as a result of their design. For example PL/I allows data declarations to be placed anywhere within a program, so no code can be generated until the entire program has been scanned. In contrast, many programming languages have been designed specifically to be compiled with one-pass compilers, and include special constructs to allow one-pass compilation.

3. How the following statement is translated via the different phases of compilation? Explain.

MOTION = DISTANCE + RATE * DISPLACEMENT + 70. [WBUT 2015]

Answer:



4. Explain in detail the process of compilation. Develop the output of each phase of the compilation for the input `a = (b + c) * (b + c) *2` [WBUT 2019]

Answer:

1st Part: ***Refer to Question No. 1 of Long Answer Type Questions.***

2nd Part:

Phase 1: Lexical Analysis

Tokens for the expression 1

Symbol	Category	Attribute
A	Identifier	#1
=	operator	Assignment(1)
B	Identifier	#2
+	operator	Arithmetic(1)
C	Identifier	#3
*	operator	Arithmetic(2)
(	operator	Open parenthesis(1)
)	operator	Closed parenthesis(1)
2	Constant	#4

Phase 2: Syntax Analysis

```
>> T1 = inttoreal (2)
>> T2 = b + c
>> T3 = T2 * T2
>> T4 = T3 * T1
>> a = T4
```

Phase 4: Intermediate Code generation

```
>> MOV R2,b
>> ADD R2,c
>> MUL R2, R2
>> MUL R2, #2.0
>> MOV R2, a
```

Phase 5: Code optimization

```
>> MOV b,R2
>> ADD R2,c
>> MUL R2, R2
>> SHL R2
>> MOV R2, a
```

5. Write short notes on the following:

a) Cross Compiler **[WBUT 2008, 2010, 2013, 2016]**

b) Chomsky classification of grammar **[WBUT 2012]**

Answer:

a) Cross Compiler:

A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is run. Cross compiler tools are used to generate executables for embedded system or multiple platforms. It is used to compile for a platform upon which it is not feasible to do the compiling, like microcontrollers that don't support an OS.

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.
- Compiling for multiple machines—a company supporting several different versions of OS, a single build environment can be set up to compile for each of these targets.
- Compiling on a server farm.
- Compiling native code for emulators.

b) Chomsky classification of grammar:

Type 0, Grammar is any phrase structure grammar without any restrictions.

To define the other types we need a definition.

In a production of the form $\phi A\psi \rightarrow \phi\alpha\ \psi$, where A is a variable, ϕ is called the left context ψ, the right context and $\phi\alpha\ \psi$ the replacement string.

Example: (a) i) In abAbcd->abABbcd, ab is the left context, bcd is the right context, α=ab

ii) In AC->A, A is the left context, ^ is the right context. α=^ The production simply erases C when the left context is A and the right context is ^.

iii) a) In C-> ^, the left and right context are ^.α=^ The production simply erases C when in any context.

Type 1, Grammars is a production of the form $\phi A\psi \rightarrow \phi\alpha\ \psi$ is called type 1 production if $\alpha \neq$ ^. In type 1 production erasing of A is not permitted.

A grammar is called ***Type 1*** or ***context sensitive*** or ***context dependent*** if all the productions are Type 1 production. The production S->^ is also allowed in a Type 1 grammar, but in this case S does not appear on the R.H.S of any production.

Example: (b) i) In aAbcD->abcDbcD is the type 1 production. a, bcD are the left context, and right context.

ii) AB -> AbBc is a type 1 production. The left context is A and right context is ^.

iii) A-> abA is a type 1 production. The both left and right context is ^.

Type 2, Grammars is a production of the form A->α, where A,B$\in V_N$ and $\alpha \in (V_N \cup \Sigma)^*$ In other words, the L.H.S has no left context or right context.

A grammar is called ***Type 2*** or ***context free grammar*** if all the productions are Type 2 productions.

Example: (c) S ->Aa, A ->a, B ->abc, A ->^, are type 2 productions.

Type 3, Grammars is a production S -> ^ is allowed in type 3 grammar, but in this case S does not appear on the right-hand side of any production.

A grammar is called ***Type 3*** or ***regular grammar*** if all the productions are Type 3 productions.

Example: (d) S ->aS, A ->a, A ->aB, are type 3 productions.

Language: A language L over an alphabet A is a collection of words on A, A^* denotes the set of all words on A. Thus a language L is simply a subset of A^*.

e.g., $\Sigma = \{a, b\}$ then

$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aab, bbb \ldots\}$

The set

$\{a, aa, aab\}$

is a language on Σ. Because it has a finite number of sentences, we call it a finite language. The set

$L = \{a^n b^n : n \geq 0\}$

is also a language on Σ. The string aabb and aaabbb are in the language L, but the string abb is not in L. This language is infinite.

LEXICAL ANALYSIS

Multiple Choice Type Questions

1. If x is a terminal then FIRST(x) is [WBUT 2007, 2008, 2010, 2012, 2014, 2016, 2017]

a) ε b) $\{x\}$ c) $x*$ d) none of these

Answer: (b)

2. The regular expression $(a|b)*abb$ denotes [WBUT 2009, 2016]

a) all possible combinations of *a*'s and *b*'s
b) set of all strings endings with *abb*
c) set of all strings starting with *a* and ending with *abb*
d) none of these

Answer: (b)

4. The following productions of a regular grammar generates a language L.
S→aS | bS | a | b
The regular expression for L is [WBUT 2009, 2015, 2016]

a) $a+b$ b) $(a+b)^*$ c) $(a+b)(a+b)^*$ d) $(aa+bb)a^*$

Answer: (c)

5. White spaces and tabs are removed in [WBUT 2011, 2018]

a) Lexical analysis b) Syntax analysis
c) Semantic analysis d) all of these

Answer: (a)

6. Which of the following is used for grouping of characters into tokens?

a) Parser b) Code optimization [WBUT 2013]
c) Code generator d) Lexical analyzer

Answer: (d)

7. ..... or scanning is the process where the stream of characters making up the source program is read from left to right grouped into tokens. [WBUT 2013, 2019]

a) Lexical analysis b) Diversion
c) Modeling d) None of these

Answer: (a)

8. The regular expression representing the set of all strings over (*x*, *y*) ending with *xx* beginning with *y* is [WBUT 2016]

a) $xx(x+y)^*y$ b) $y(x+y)^*xx$ c) $yy(x+y)^*x$ d) $y(xy)^*xx$

Answer: (b)

9. The basic limitation of Finite State Machine is that **[WBUT 2016]**

a) it cannot remember arbitrary large amount of information
b) it cannot recognize grammars that are regular
c) it sometimes recognize grammars that are not regular
d) all of these

Answer: (a)

10. What is the output of lexical analyzer? **[WBUT 2018]**

a) A parse tree b) A list of tokens c) A syntax tree d) None of these

Answer: (b)

11. Regular expression (*x*/*y*) (*x*/*y*) denotes the set **[WBUT 2019]**

a) {*xy*, *xy*} b) {*xx*, *xy*, *yx*, *yy*} c) {*x*, *y*} d) {*x*, *y*, *xy*}

Answer: (b)

12. The regular expressions denote zero or more instances of an *x* or *y* is **[WBUT 2019]**

a) (x + y) b) (x + y)* c) (x* + y) d) (xy)*

Answer: (b)

Short Answer Type Questions

1. What do you understand by terminal table and literal table? **[WBUT 2006, 2008, 2010, 2011]**

Answer:

During lexical analysis, after a token is identified, first the terminal table is examined. If the token is not found in the table, a new entry is made. Only the 'name' attribute of the token goes in, the remaining information is inserted in later phases. On the other hand, numbers, quoted character strings and other self-defining data are classified as 'literals' and go to the literal table. If the literal is not found in the table, a new entry is made. The lexical analyzer can determine all the attributes of a literal as well as its internal representation, by looking at it.

2. What is a lookahead operator? Give an example. With the help of the look ahead concept show how identifiers can be distinguished from keywords. **[WBUT 2007, 2009, 2016]**

Answer:

For certain programming languages, the lexical analyzers may have to look ahead a few symbols beyond the end of a lexeme before they can determine a token with certainty. In LEX, the lookahead operator is '/'.

For example, in Fortran-77, the token IF can be the keyword "IF" or it can be the name of a function/array as in IF(10, 5). It is the keyword only if a pair of parentheses has been seen, followed by a letter. Hence, in a Lexical

Analyzer for Fortran-77, we would specify the keyword IF as:

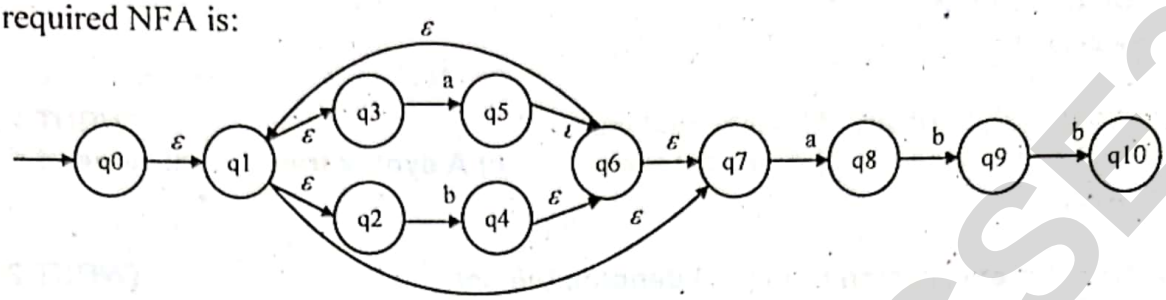
IF / : {letter} where we assume that the definition {letter} means any letter.

3. Give the NFA for the following Regular Expression. Then find a DFA for the same language.

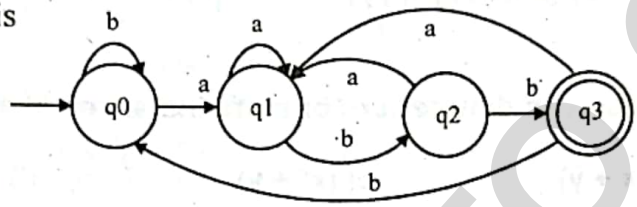
(a|b) * abb **[WBUT 2008, 2013]**

Answer:

The required NFA is:



The equivalent DFA is

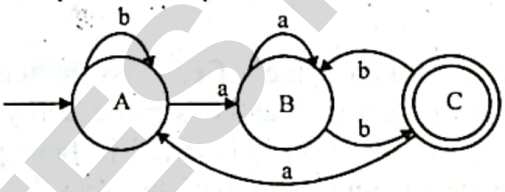


4. Construct DFA directly from [not by generating NFA] the regular expression

$L = (a|b)^* ab$ **[WBUT 2010, 2011, 2018]**

Answer:

The required DFA for the language $L = (a|b)^* ab$ is as given below:

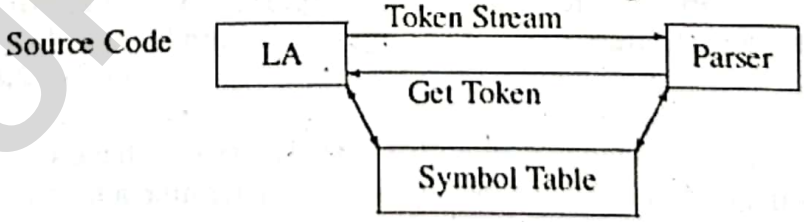


5. Describe with diagram the working process of lexical Analyzer.

[WBUT 2011, 2014, 2018]

Answer:

The operation of the Lexical Analyzer is explained in the diagram below.



The Lexical Analyzer breaks down the source program into tokens like keywords, constants, identifiers, operators and other simple tokens. A token is the smallest piece of text that the language defines.

Keywords are words the language defines, and which always have specific meaning in the language like **if, else, int, char, do, while, for, struct, return**, etc in C/C++.

Constants are the literal valued items that the language can recognize, like numbers, strings, and characters.

Identifiers are names the programmer has given to something. These include variables, functions, classes (in C++ and other object-oriented languages), enumerations, etc. Each language has rules for specifying how these names can be written.

Operators are the mathematical, logical, and other operators that the language can recognize.

For identifiers, the Lexical Analyze also uses a "Symbol Table" — a data structure it shares with the Parser and subsequent phases of compilation.

6. Define regular expression. Write the regular expression over alphabet $\{a,b,c\}$ containing at least one 'a' and at least one 'b'. What is dead state? Explain with suitable example. **[WBUT 2012]**

Answer:

A regular expression (RE) is defined recursively as follows:

ε is an RE

a is an RE, where '$a' \in \varepsilon$

If R_1 and R_2 are REs, then so are

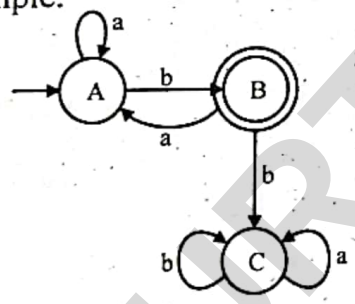
$R_1 + R_2$, $R_1 R_2$ and R^*.

Required RE is:

$$\left(\left((a+b+c)^* a(a+b+c)^* b(a|b|c)^*\right)\left((a+b+c)^* b(a+b+c)^* a(a+b+c)^*\right)\right)$$

In a finite state automata; a dead state is one such that once that state is reached, any input keeps the automata in that state is reached, any input keeps the automata in that state.

Example:



Here, C is a dead-state.

7. How does lexical analyzer help in the process of compilation? Consider the following statement and find the number of tokens with type and value as applicable:

```
void main ( )
    {
        int x;
        x = 3;
    }
```

[WBUT 2012]

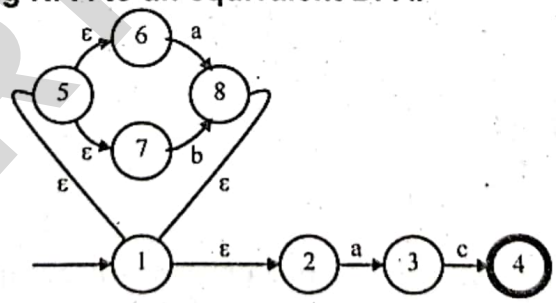
Answer:
The lexical analyzer breaks down the source code into a stream of 'tokens'. The grammar of a language is expressed in terms of tokens where similar types of entities are treated similarly. For examples, variables and constants are considered as "identifiers". This helps in the compiler being based on a language of token types, which is not dependent on the particular source code being compiled.
There are total 13 tokens:
Void : Keyword
main : identifier
(
)
{
int : keyword
x : identifier
;
x
=
3 : identifier
;
}

8. What is Token, Pattern and Lexeme? **[WBUT 2013]**

Answer:
Lexeme: group of characters that form a token
Token: class of lexemes that match a pattern
- Token may have attributes, if more than one lexeme in token

Pattern: typically defined using a regular expression
- REs are simplest language class that's powerful enough

9. Convert the following NFA to an equivalent DFA. **[WBUT 2015]**



Answer:
To Remove Epsilon moves we follow following steps:
- Find Closure of all states which have null moves
- Mark these states which have null moves
- Make a revised transition table without epsilon column and Find all possible transition for those marked states by using their Closures.
- We will get nfa without null moves/epsilon moves

Initial transition table:

States	a	b	c	epsilon
Q_1				$\{Q_2, Q_5\}$
Q_2	Q_3			
Q_3			Q_4	
Q_4				
Q_5				$\{Q_6, Q_7\}$
Q_6	Q_8			
Q_7		Q_8		
Q_8				Q_1

Finding Closure of Q_1, Q_5, Q_8

Closure $(Q_1) = \{Q_1, Q_2, Q_5, Q_6, Q_7\}$ Closure $(Q_5) = \{Q_5, Q_6, Q_7\}$

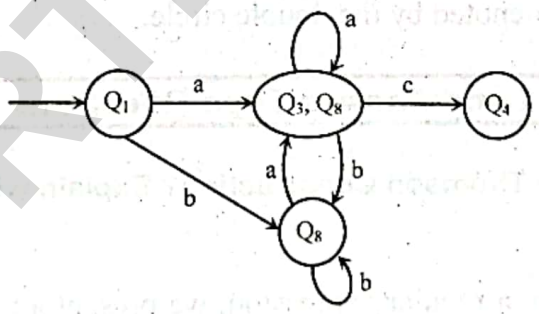
Closure $(Q_8) = \{Q_1, Q_8\}$

We find a transition table after following above steps which is:

States		**Input**	
	a	**b**	**c**
Q_1	Q_3, Q_8	Q_8	
Q_3			Q_4
Q_8	Q_3, Q_8	Q_8	
Q_4	*		

So if we change the nfa to a dfa the transition table and the corresponding dfa will be as follows:

States		**Input**	
	a	**b**	**c**
Q_1	Q_3, Q_8	Q_8	
Q_3, Q_8	Q_3, Q_8	Q_8	Q_4
Q_8	Q_3, Q_8	Q_8	
Q_4	*		



10. Find out regular expression corresponding to the finite automata: [WBUT 2016]

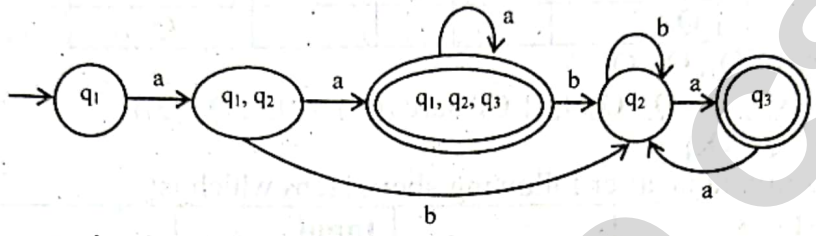
PS	Next State	Next State
	a	b
q1	q1, q2	–
q2	q3	q2, q2
q3	q2	–

Answer:

The DFA table is

Present State	Next state on a	Next state on b
q1	q1, q2	–
q2	q3	q2
(q1 , q2)	(q1, q2, q3)	q2
(q1, q2, q3)	(q1, q2, q3)	q2
q3	q2	–

If we consider q3 as final state then the state diagram be like below



So the regular expression is a*b*(a a b*)*a.

11. Define NFA. **[WBUT 2019]**

Answer:

A Non- deterministic Finite Automata or NFA has five states (Q, ∑, q0, F, δ) where

1. Q: finite set of states
2. ∑: finite set of the input symbol
3. q0: initial state
4. F: final state
5. δ: Transition function

An NFA can be represented by digraphs called state diagram. In which:

(i) The state is represented by vertices.
(ii) The arc labeled with an input character show the transitions.
(iii) The initial state is marked with an arrow.
(iv) The final state is denoted by the double circle.

Long Answer Type Questions

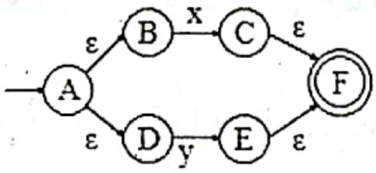
1. What do you mean by Thomson Construction? Explain with an example. **[WBUT 2015]**

Answer:

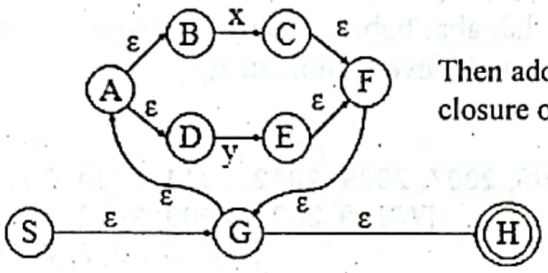
To construct an NFA from a regular expression, we present a technique that can be used as a recognizer for the tokens corresponding to a regular expression. In this technique, a regular expression is first broken into simpler subexpression, then the corresponding NFA are constructed and finally, these small NFAs are combined with the help of regular expression operations. This construction is known as **Thompson's construction.**

Thompson Construction Example:

Develop an NFA for the RE: (x|y)*



First create NFA for (x|y)



Then add in the closure operator

2. What is Regular Expression? Write the regular expression for: [WBUT 2019]

a) R = R1 + R2 (Union operation)

b) R = R1.R2 (Concatenation operation)

c) R = R1* (Kleen Clouser)

d) R = R + (Positive Clouser)

e) Write a regular expression for a language containing strings which end with "abb" over ∑ = {a, b}.

f) Construct a regular expression for the language containing all strings having any number of *a*'s and *b*'s except the full string.

Answer:

1st Part: ***Refer to Question No. 6(1st Part) of Short Answer Type Questions.***

2nd Part:

a) If R1 and R2 are regular expressions, then R1 | R2 (also written as R1 U R2 or R1 + R2) is also a regular expression.
L(R1|R2) = L(R1) U L(R2).

b) If R1 and R2 are regular expressions, then R1R2 (also written as R1.R2) is also a regular expression.
L(R1R2) = L(R1) concatenated with L(R2).

c) If R1 is a regular expression, then R1* (the Kleene closure of R1) is also a regular expression.
L(R1*) = epsilon U L(R1) U L(R1R1) U L(R1R1R1) U ...
Closure has the highest precedence, followed by concatenation, followed by union.

d) RS is a regular expression whose language is L, M. R+ is a regular expression whose language is L+.

e) The Regular expression for a language containing strings which end with "abb" is:
R. E. = (a + b)* abb
A set of strings of a's and b's ending with the string abb. So L = {abb, aabb, babb, aaabb, ababb, ……..}.

f) The regular expression for the language which containing all strings having any number of a's and b's except the full string is:

Regular Expression (R.E) = $(a + b)^*$.

This will give the set as L = {ε, a, aa, b, bb, ab, ba, aba, bab, ...} any combination of a and b. The (a + b)* shows any combination with a and b even a null string.

3. Write short notes on the following:

a) YAAC **[WBUT 2006, 2007, 2009, 2012, 2013, 2014, 2016]**

b) LEX **[WBUT 2007, 2009, 2012, 2013]**

c) Thompson's Construction Rule **[WBUT 2007, 2014]**

d) LEX and YAAC **[WBUT 2010, 2011, 2018]**

Answer:

a) YAAC:

yacc assumes that the user has supplied a lexical analyzer which is an integer-valued function called yylex. The function returns an integer, the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable yylval. The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place.
In normal practice, one uses LEX to generate the lexical analyzer which is then piggybacked into the parser code generated by yacc.

b) LEX:

Lex is a program that generates lexical analyzers. Lex is commonly used with the yacc parser generator. Lex is the standard lexical analyzer generator on many Unix systems. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexical analyzer in the C programming language.
The structure of a lex file is intentionally similar to that of a yacc file; files are divided up into three sections, separated by lines that contain only two percent signs, as follows:
Definition section
%%
Rules section
%%
C code section
The definition section is the place to define macros and to import 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 is the most important section; it associates patterns with C statements. Patterns are simply regular expressions. When the lexer sees some text in the input matching a given pattern, it executes the associated C code. This is the basis of how lex operates. The C code section contains C statements and functions

that are copied verbatim 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 and link it in at compile time.

c) Thompson's Construction Rule:

Given any RE, it is possible to algorithmically construct an NFA such that the language accepted by the NFA is exactly the language expressed by the RE.

This is done by systematically using the constructs for the basic primitives and operators of an RE and building ups using Thompson's Construction process. The steps are:

Step-1, Parse the given RE into its constituent sub-expressions.

Step-2, Construct NFA-s for each of the basic symbols in the given RE.

For ε, construct the NFA as in Fig-1. Here i is a new start state and f is a new accepting state. It is clear that this NFA recognizes the RE {ε}.

For every a in the alphabet S, construct the NFA as shown in Fig-2. Here again i is a new start state and f is a new accepting state. It is clear that this NFA recognizes the RE {a}. If a occurs several times in the given RE, a separate NFA is constructed for each occurrence of a.

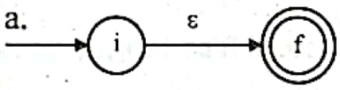


Fig 1. NFA for e

Fig 2. NFA for a character of the alphabet

Step-3, Combine the intermediate NFA-s inductively until the NFA for the entire expression is obtained. Each intermediate NFA produced during the course of construction corresponds to a sub-expression of the given RE and has several important properties – it has exactly one final state, no edge enters the start state and no edge leaves the final state.

Suppose N(s) and N(t) are the NFA-s for regular expressions s and t, respectively. The NFA-s for regular expression s|t, st and s*, the composite NFA-s N(s|t), N(st) and N(s*) are constructed as shown in

Fig- 3, Fig-4 and Fig-5, respectively:

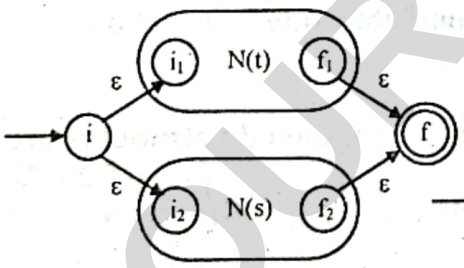


Fig: 3

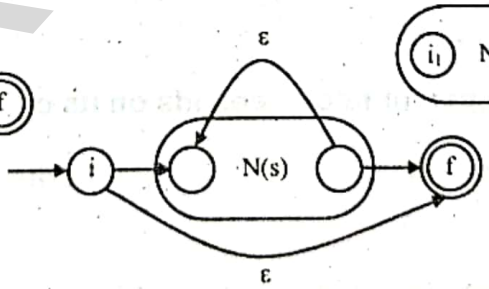


Fig.:5

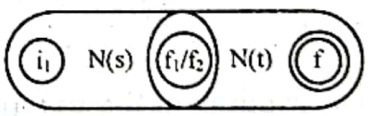


Fig: 4

d) LEX: ***Refer to Question No. 3(a) of Long Answer Type Questions.***

YAAC: ***Refer to Question No. 3(b) of Long Answer Type Questions.***

PARSING AND CONTEXT FREE GRAMMAR

Multiple Choice Type Questions

1. If all productions in a grammar G = (V,T,S,P) are of the form A→xB or A→x, A,B ∈ V and x ∈ T*, then it is called: [WBUT 2008]
   a) Context-sensitive grammar
   b) Non-linear grammar
   c) Right-linear grammar
   d) Left-linear grammar

Answer: (c)

2. An inherited attributes is the one whose initial value at a parse tree node is defined in terms of [WBUT 2009, 2015]
   a) attributes at the parent and / or siblings of that node
   b) attributes at children nodes only
   c) attributes at both children nodes and parent and / or siblings of that node
   d) none of these

Answer: (c)

3. The intersection of a regular language and a context free language is [WBUT 2009]
   a) always a regular language
   b) always a context free language
   c) always a context sensitive language
   d) none of these

Answer: (b)

4. The grammar E →E+E | E*E | α is [WBUT 2010, 2013]
   a) ambiguous
   b) unambiguous
   c) not given sufficient information
   d) none of these

Answer: (a)

5. Parse tree is generated in the phase of [WBUT 2011, 2018]
   a) Syntax Analysis
   b) Semantic Analysis
   c) Code Optimization
   d) Intermediate Code Generation

Answer: (a)

6. If the attributes of the parent node depends on its children, then its attributes are called [WBUT 2012]
   a) TAC
   b) synthesized
   c) inherited
   d) directed

Answer: (c)

7. The expression wcw where w belongs to {a, b} * is [WBUT 2014]
   a) regular
   b) context free
   c) context sensitive
   d) none of these

Answer: (b)

8. The grammar $S \rightarrow S\alpha_1 | S\alpha_2 | \beta_1 | \beta_2$ [WBUT 2015]
   a) is left recursive
   b) has common left factor
   c) is left recursive and also has common left factor
   d) is a CFG

Answer: (a)

9. An annotated parse tree is a parse tree [WBUT 2016]
   a) with values of only some attributes shown at parse tree nodes
   b) with attribute values shown at the parse node
   c) without attribute values shown at the parse tree nodes
   d) with grammar symbols at the parse tree nodes

Answer: (b)

10. The output of the parser is [WBUT 2017]
   a) tokens b) syntax tree c) parse tree d) non-terminals

Answer: (c)

11. The grammar S → aSa | bS | c is [WBUT 2018]
   a) LL(1) but not LR (1) b) LR(1) but not LL(1)
   c) Both LL(1) and LR(1) d) None of these

Answer: (c)

12. Grammar of the programing is checked at..... phase of compiler. [WBUT 2019]
   a) Semantic analysis b) Syntax analysis
   c) Code optimization d) Code generation

Answer: (b)

13. A grammar that produce more than one parse tree [WBUT 2019]
   a) Ambiguous b) Unambiguous c) Regular d) All of these

Answer: (a)

14. ..... is the most general phase structured grammar. [WBUT 2019]
   a) Context sensitive b) Regular
   c) Context tree d) All of these

Answer: (a)

Short Answer Type Questions

1. What is 'handle'? Consider the grammar E→E +n|E *n|n. For a sentence n+n*n, write the handles in the right-sentential forms of the reduction.
What is predictive parsing? [WBUT 2006]

OR,

What do you mean by a Handle? Give example. [WBUT 2013]

Answer:
A handle of a right sentential form gamma is a production A!b and a position in gamma where the string beta may be found and replaced by A to get the previous right-sentential form.
The right-most productions are shown below. The handles are underlined.

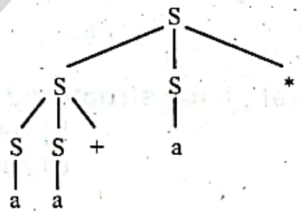
E→E*n→E+n*n→n+n*n

The handles are underlined.
A predictive parser is a recursive descent parser that does not require backtracking. Predictive parsing is possible only for the class of LL(k) grammars, which are the context-free grammars for which there exists some positive integer k that allows a recursive descent parser to decide which production to use by examining only the next k tokens of input. (The LL(k) grammars therefore exclude all ambiguous grammars, as well as all grammars that contain left recursion. Any context-free grammar can be transformed into an equivalent grammar that has no left recursion, but removal of left recursion does not always yield an LL(k) grammar.) A predictive parser runs in linear time.
A table-driven non-recursive predictive parser (for an LL(1) grammar) uses an explicit parsing stack and a parsing table M[A;a] (where A is a nonterminal and a is a terminal) where an entry M[A:a] is either a production rule or an error. A predictive parsing algorithm uses this table to carry out top-down parsing.

2. Consider the following context-free grammar: **[WBUT 2008, 2009]**
S → SS+ | SS* | a
a) Show how the string aa+a* can be generated by this grammar.
b) Construct a parse tree for the given string.
c) What language is generated by this grammar?
Answer:
a) The string aa+a* can be generated by the rightmost production:

$$S \rightarrow SS^* \rightarrow Sa^* \rightarrow SS + a^* \rightarrow Sa + a^* \rightarrow aa + a^*$$

b) The required parse tree is:



c) This grammar generates Binary Postfix Expressions involving a with + and * as the only operators.

3. Consider the following left-linear grammar: **[WBUT 2008]**

$$S \rightarrow Sab \mid Aa$$

$$A \rightarrow Abb \mid bb$$

Find out an equivalent right-linear grammar.

Answer:

This grammar generates the regular language (bb)+a(ab)+ where + means "one or more occurrences". An equivalent right-linear grammar is:

$S \rightarrow bbS \mid bbaA$

$A \rightarrow abA \mid ab$

4. What is a handle? **[WBUT 2008, 2016, 2017]**

Consider the grammar $E \rightarrow E+E \mid E*E \mid id$

Find the handles of the right sentential forms of reduction of the string $id+id*id$

Answer:

A handle of a right sentential form gamma is a production A—>b and a position in gamma where the string beta may be found and replaced by A to get the previous right-sentential form.

The given grammar is ambiguous. There are two right-most derivations for the string id + id * id. We give both the rightmost derivations, underlining the handles in each case.

$E \rightarrow E+E \rightarrow E+E*E \rightarrow E+E*id \rightarrow E+id*id \rightarrow id+id*id$

$E \rightarrow E*E \rightarrow E*id \rightarrow E+E*id \rightarrow E+id*id \rightarrow id+id*id$

5. What is error handling? Describe the Panic Mode and Phrase level error recovery technique with example. **[WBUT 2011, 2018]**

Answer:

Programs submitted to a compiler often have errors of various kinds. When a compiler detects an error, i.e., when the symbols in a sentence do not match the compiler's current position in the syntax diagram, the error handler is invoked. The error handler warns the programmer by issuing an appropriate message and then attempts to recover from it so that it can detect more errors.

The simplest form of error recovery technique is "panic mode". A small set of 'safe symbols' are used to delimit 'clean points' in the input. When an error occurs, a panic mode recovery algorithm deletes input tokens until it finds a safe symbol, then backs the parser out to a context in which that symbol might appear. In the following fragment of Pascal code: if a b then x else y;

Compiler discovers the error at b and a panic-mode recovery algorithm very likely skips forward to the semicolon, thereby missing the then. When the parser later finds the else, it again produces a spurious error message.

In phrase-level error recovery, the quality of recovery is improved by employing different sets of safe symbols in different contexts. When compiler discovers an error in an expression, it deletes input tokens until it reaches something that is likely to follow an expression. This more local recovery is better than always backing out to the end of the current statement because it gives the compiler the opportunity to examine the parts of the statement that follow the erroneous expression. In our above example, a phrase level recovery would use the then and else tokens as the next safe symbols and give a more realistic error message.

6. Consider the following grammar G. Alternate the production so that it may free from backtracking.
Statement → if Expression then Statement else Statement
Statement → if Expression then Statement **[WBUT 2012]**

Answer:
Statement → if Expression then Statement Trailer
Trailer → else Statement
Trailer → ε

7. Explain left factoring with suitable example. **[WBUT 2013, 2014]**

Answer: ***Refer to Question 5(a) of Long Answer Type Questions.***

8. Consider the context-free grammar: **[WBUT 2017]**

$S \to SS+ \mid SS^* \mid \varepsilon$

a) How the string $aa+a^*$ can be generated by this grammar?
b) Construct a parse tree for this string.

Answer:

Similar to Question No. 2(a) of Short Answer Type questions.

9. Eliminate the left-recursion for the following grammar: **[WBUT 2017]**

$S \to (L) \mid a$
$L \to LS \mid S$

Answer:

$$\left.\begin{array}{l} S \to (L) \mid a \\ \boxed{L \to L\underline{S} \mid \underline{S}} \end{array}\right\} \text{Left recursion elimination}$$

$\alpha \quad \beta$

$\downarrow \qquad \downarrow$

$$\boxed{\begin{array}{l} L \to SL' \\ L' \to SL' \mid \varepsilon \end{array}} \qquad \boxed{\begin{array}{l} A \to A\alpha \mid \beta \\ A \to \beta A' \\ A' \to \varepsilon \mid \alpha A' \end{array}} \to \text{(form of left recursion)}$$

$\Rightarrow$ After left recursion elimination →

$$\left.\begin{array}{l} S \to (L) \mid a \\ L \to SL' \\ L' \to SL' \mid \varepsilon \end{array}\right\} \quad \textbf{(Ans.)}$$

Long Answer Type Questions

1. Construct a predictive parsing table for the grammar: **[WBUT 2010]**

S → iEtSS′ | a
S′ → eS | ε
E → b

Here S is star symbol and S′ are non-terminals and i, t, a, e, b are terminals. Explain the steps in brief.

Answer:

From rules $S \to iEtSS' \mid a$, we get $FIRST(S) = \{i, a\}$.

From rules $S' \to eS; \mid \varepsilon$, we get $FIRST(S') = \{e, \varepsilon\}$.

From rule $E \to b$, we get $FIRST(E) = \{b\}$.

Using the FIRST sets and the rules, we get:

$FOLLOW(S) = \{\$\} \cup FIRST(S') = \{e, \$\}$.

$FOLLOW(S') = \{e, \$\}$.

$FOLLOW(E) = \{t\}$.

Suppose the predictive parsing table is the 2-dimensional array $P[A, a]$, where A is a non-terminal and a is a terminal.

Since $i \in FIRST(S)$, $P[S, i] = S \to iEtSS'$.

Since $a \in FIRST(S)$, $P[S, a] = S \to a$.

Since $e \in FIRST(S')$, $P[S', e] = S' \to eS$. Also, since $\varepsilon \in FIRST(S')$, $P[S', e] = S' \to \varepsilon$ because $e \in FOLLOW(S')$. Hence, we have multiple entries for $M[S', e]$. Since $\$ \in FOLLOW(S')$ and obviously $\varepsilon \in FIRST(\varepsilon)$, we have $P[S', \$] = S' \to \varepsilon$.

Since $b \in FIRST(E)$, $P[E, b] = E \to b$.

No other rule is left to be scanned. So the predictive parsing table is:

Non-terminal	Input Symbol					
	i	t	e	a	b	$
S	$S \to iEtSS'$			$S \to a$		
S'			$S' \to eS$ $S' \to \varepsilon$			$S \to \varepsilon$
E					E → b	

2. a) Define LL(1) grammar. Consider the following grammar: **[WBUT 2012]**

$S \to AaAb \mid BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

Test whether the grammar is LL(1) or not and construct a predictive parsing table for it.

Answer:

FIRST (A) = $\{\varepsilon\}$, FIRST (B) = $\{\varepsilon\}$

FIRST (S) = $\{a, b\}$

FOLLOW (A) = $\{a, b\}$, FOLLOW (B) = $\{a, b\}$, FOLLOW (S) = $\{\$\}$

The predictive parsing table is:

	a	b	$
S	S → AaAb	S → BbBa	
A	A → ε	A → ε	
B	B → ε	B → ε	

Since there are no conflicts, the grammar is LL(1)

b) Consider the following Context Free Grammar (CFG) G and reduce the grammar by removing all unit productions. Show each step of removal [WBUT 2012]

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow C|b$

$C \rightarrow D$

$D \rightarrow E$

$E \rightarrow a$

Answer:

State : S → AB
A → a
B → C | b
C → D
D → E
E → a

Step-1: S → AB
A → a
B → C | b
C → D
D → a
E → a

Step-2: S → AB
A → a
B → C | b
C → a
D → a
E → a

Step-3: $S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow a \mid b$
$C \rightarrow a$
$D \rightarrow a$
$E \rightarrow a$

c) Consider the following grammar G. Show that the grammar is ambiguous by constructing two different leftmost derivations for the sentence '*abab*'.

$S \rightarrow aSbS \mid bSaS \mid \varepsilon$ **[WBUT 2012]**

Answer:

The two possible left-most derivations for 'abab' are:

(i) $S \xrightarrow{S \rightarrow aSbS} a\underline{S}bS \xrightarrow{S \rightarrow \varepsilon} ab\underline{S} \xrightarrow{S \rightarrow aSbS} aba\underline{S}bS \xrightarrow{S \rightarrow \varepsilon} ababS \xrightarrow{S \rightarrow \varepsilon} abab$

(ii) $S \xrightarrow{S \rightarrow aSbS} a\underline{S}bS \xrightarrow{S \rightarrow bSaS} ab\underline{S}aSbS \xrightarrow{S \rightarrow \varepsilon} aba\underline{S}bS \xrightarrow{S \rightarrow \varepsilon} ababS \xrightarrow{S \rightarrow \varepsilon} abab$

3. a) Consider the grammar: **[WBUT 2014]**

$S \rightarrow aSbS \mid bSaS \mid E$

(i) Show that this grammar is ambiguous by constructing two different left most derivations for the sentence abab.
(ii) Construct the corresponding right most derivations for abab.
(iii) Construct the corresponding parse trees for abab.
(iv) What language does this grammar generate?

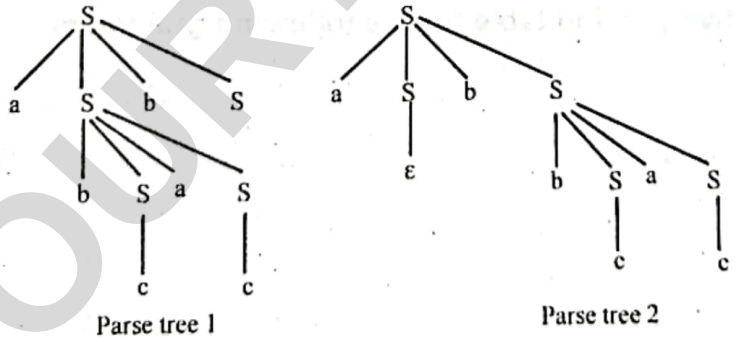
b) (i) Show that no left recursive grammar can be LL (1).
(ii) Show that no LL(1) grammar can be ambiguous.

Answer:

a) (i) ***Refer to Question No. 2(c) of Long Answer Type Questions.***

(ii) $S \rightarrow aSbS \rightarrow aSb \rightarrow abSaSb \rightarrow abSab \rightarrow abab$

(iii) Consider a string 'abab'. We can construct parse trees for deriving 'abab'.



Parse tree 1 Parse tree 2

(iv) This grammar generates the language which contains strings of equal number of a's and b's the empty string also.

b) (i) The production is left-recursive if the leftmost symbol on the right side is the same as the non terminal on the left side. For example,

expr → expr + term.

Left-recursive grammars can never be LL(1), because the left-recursion will lead to an infinite loop.

Consider the grammar

$$A \to \beta | A\alpha$$

and try to parse the string $\beta\,\alpha$.

- If we removed the left recursion the grammar becomes

  $$A \to \beta A'$$

  $$A' \to \alpha A' | \in$$

  which derives the same string but towards the right instead of the left
- Now parse $\alpha\,\alpha.\beta$ doesn't match α, therefore try another alternative for A. There are none, so parse fails. With right recursion, we will be matching part of the input string as we go along
- Left recursion indicates we are building the string from right-to-left
- To eliminate left recursion, we turn it into right recursion and build the string left-to-right

(ii) Any LL(1) grammar is unambiguous because by definition there is a at most one left most derivation for any string. LL(1) grammar cannot be ambiguous since our parsing algorithm for LL(1) grammars builds the only possible parse tree for a sentence in a deterministic way. (In general, an ambiguity in a grammar will manifest itself in the fact that there are two productions competing for the same cell in the parse table.)

4. a) Construct NFA from the regular expression using Thompson's method $L = aa(a|b)^*ab$.

b) Write regular definition for the following language:
All strings of letter that contain the five vowels in order.

c) Construct the predictive parsing table for the following grammars:

$S \to AaAb \,|\, BbBa$

$A \to \varepsilon$

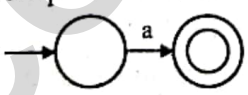
$B \to \varepsilon$

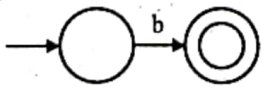
[WBUT 2017]

Answer:

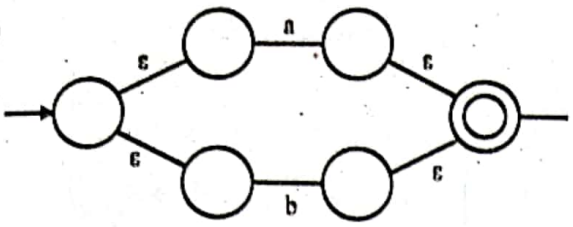
a) $L = aa(a|b)^*ab$

Thompson method →

i) 

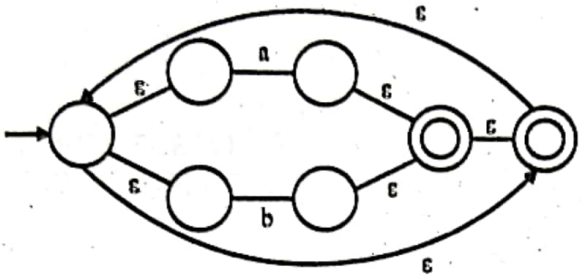
ii) 

iii)



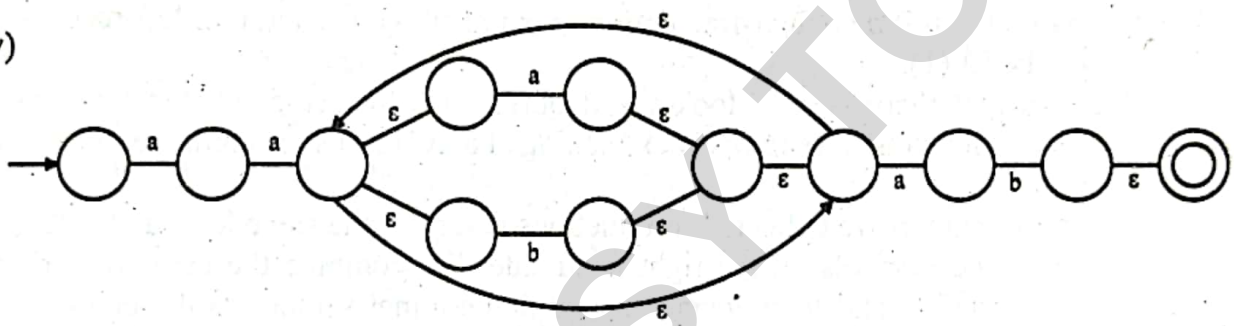
$(a|b)$

iv)



$(a|b)^*$

v)



$aa(a|b)^*ab$

b) $\sum$ = {set of all alphabets}, $\sum_c$ = {set of all consonants}

Regular Exp:

$$\sum_c^* a \sum_c^* e \sum_c^* i \sum_c^* o \sum_c^* u$$

where $\sum_c^*$ means all possible combination of alphabets (consonants).

c) Predictive Parsing Table:

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

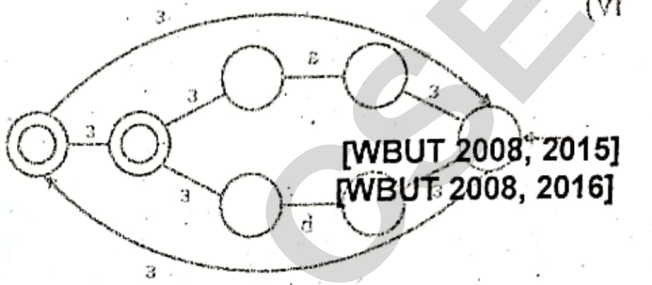
	First	Follow
$S \to AaAb$	$\{a, b\}$	$\{\$\}$
$S \to BbBa$	$\{a, b\}$	$\{\$\}$
$A \to \varepsilon$	$\{\varepsilon\}$	$\{a, b\}$
$B \to \varepsilon$	$\{\varepsilon\}$	$\{a, b\}$

(iii)

Parsing Table

	a	b	ε
S	$S \to AaAb$	$S \to AaAb$	
A	$A \to \varepsilon$	$A \to \varepsilon$	
B	$B \to \varepsilon$	$B \to \varepsilon$	

(iv)



5. Write short note on the following:
a) left factoring **[WBUT 2008, 2015]**
b) context-free grammar **[WBUT 2008, 2016]**

Answer:

a) left factoring:

Left factoring is an important step required to transform a given grammar to one that is suitable for building an LL (i.e., top-down) parser. This step is carried out after removing all left recursion. Even if a context-free grammar is unambiguous and non-left-recursion, it still may not be LL(1).

The problem is that there is only look-ahead buffer. The parser generated from such grammar is not efficient as it requires backtracking. To avoid this problem, we left factor the grammar.

To left factor a grammar, we collect all productions that have the same left hand side and begin with the same symbols on the right hand side. We combine the common strings into a single production and then append a new nonterminal symbol to the end of this new production. Finally, we create a new set of productions using this new nonterminal for each of the suffixes to the common production.

Suppose we have production rules:

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots\ldots \mid \alpha\beta_n$$

After left factoring the above grammar is transformed into

$$A \to \alpha A_1 \; A_1 \to \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$$

The above grammar is correct and is free from conflicts.

b) context-free grammar:

The formalism of Context-free grammars was developed in the mid-1950s by Noam Chomsky who used in the study of human languages (i.e., Natural Languages). Later, Context-free Grammars found an extremely important application in the specification and compilation of programming languages.

A grammar for a programming language is the starting point in the design of compilers and interpreters for programming languages. Most compilers and interpreters contain a component called a parser that extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution. A number of methodologies

facilitate the construction of a parser once a Context-free grammar is available. Some tools even automatically generate the parser from the grammar.

In terms of generative power, Context-free Grammars, or CFG-s, are more expressive than Regular Grammars (or equivalent formalisms like Regular Expression and Finite Automata).

Formally, a Context-free Grammar is defined as a 4-tuple $G = \langle V_n, V_t, P, S \rangle$, where:

- V_t is a finite set of terminals. The set of terminals constitute the alphabet S for the language as well as e.
- V_n is a finite set of non-terminals. The non-terminals constitute a special alphabet that is disjoint from the terminals.
- P is a finite set of production rules of the type $V_n \rightarrow (V_n \cup V_t)^*$.
- S is an element of V_n, the distinguished starting non-terminal, often call the Start Symbol.

OPERATOR PRECEDENCE PARSING

Multiple Choice Type Questions

1. Which data structure is mainly used during shift-reduce parsing? [WBUT 2010, 2012]

a) Pointers b) Arrays c) Stacks d) Queues

Answer: (c)

2. In operator precedence parsing, precedence relations are defined [WBUT 2013, 2019]

a) for all pair of non-terminals b) for all pair of terminals
c) to delimit the handle d) only for a certain pair of terminals

Answer: (d)

Short Answer Type Questions

1. Define an operator grammar. [WBUT 2006]

OR,

What is an operator grammar? Give an example. [WBUT 2009]

Answer:

A grammar is called an Operator Grammar if there is no production and no right hand side of any production has two adjacent non-terminals.

For example, the grammar:

$S \rightarrow x|y|z|S + S|S - S|S * S|S / S|(S)$

is an operator grammar while the grammar:

$S \rightarrow ASA$

$A \rightarrow a|b$

is not an operator grammar since the rule $A \rightarrow ASA$ has two non-terminals side by side.

2. Describe the algorithm for eliminating left recursion from a CFG. Eliminate left recursion from the following grammar.

$S \rightarrow Aa|b$

$A \rightarrow Ac|Sd|\varepsilon$ [WBUT 2015]

Answer:

1st Part:

Algorithm for Eliminating General Left Recursion

```
Arrange nonterminals in some order A1, A2, ... , An.
for i :=1 to n do begin
    for j := 1 to i-1 do begin
      Replace each production of the form Ai -> Ajβ
      by the production:
       Ai → α2β|α2β| ... |αkβ
```

```
      where
       Aj → α2 | α2 | ... | αk
      are all the current Aj productions.
    end ( for j )
    Remove immediate left recursion from the Ai
    productions, if necessary
end   ( for i )
```

2nd Part:

- Let's use the ordering S, A ($S = A_1$, $A = A_2$).
- When i = 1, we skip the "for j" loop and remove immediate left recursion from the S productions (there is none).
- When i = 2 and j = 1, we substitute the S-productions in $A \rightarrow Sd$ to obtain the A-productions

  $A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$
- Eliminating immediate left recursion from the A productions yields the grammar:

  $S \rightarrow Aa \mid b$

  $A \rightarrow bdA' \mid A'$

  $A' \rightarrow CA' \mid adA' \mid \varepsilon$

3. Describe about the operator precedence parser. **[WBUT 2017]**

Answer:

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has a∈.
- No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".

a ⋖ b means that terminal "a" has the lower precedence than terminal "b".

a ≐ b means that the terminal "a" and "b" both have same precedence.

Parsing Action

- Both end of the given input string, add the $ symbol.
- Now scan the input string from left right until the ⋗ is encountered.
- Scan towards left over all the equal precedence until the first left most ⋖ is encountered.
- Everything between left most ⋖ and right most ⋗ is a handle.
- $ on $ means parsing is successful.

	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	X
)	>	>	X	>	X	>
id	>	>	X	>	X	>
$	<	<	<	X	<	X

4. Remove left recursion $S \rightarrow Aa|b$, $A \rightarrow Ac|Sd|e$. [WBUT 2019]

Answer:

Step-1: Remove immediate left recursion in $A \rightarrow Ac$

$A \rightarrow SdA' | eA'$

$A' \rightarrow cA' | \varepsilon$

Step-2:

(i) $S \rightarrow SdA'a | cB'a | b$

$A' \rightarrow SdA' | eA'$

$A' \rightarrow cA' | \varepsilon$

(ii) After removing immediate left recursion is

$S \rightarrow SdA'a$

$S \rightarrow eA'aS' | bS'$

$S' \rightarrow dA'aS' | \varepsilon$

$A \rightarrow SdA' | eA'$

$A' \rightarrow cA' | \varepsilon$

The grammar is now free from left recursion.

Long Answer Type Questions

1. a) What is left-recursion? Illustrate with suitable example. Consider the following grammar G. Find out the left recursion and remove it: [WBUT 2012]

$S \rightarrow Bb|a$

$B \rightarrow Bc|Sd|e$

Answer:

1st Part:

Left recursion: when one or more productions can be reached from themselves with no tokens consumed in-between. *Left recursion* is a particular form of recursion that cannot be directly handled by the simple LL (1) parsing algorithm. Left recursion just refers to any recursive non-terminal that, when it produces a sentential form containing itself, that new copy of itself appears on the left of the production rule.

2nd Part:

Step-1: Remove immediate left recursion in $B \rightarrow Bc$

$B \rightarrow SdB' \mid eB'$

$B' \rightarrow cB' \mid \varepsilon$

Step-2:

(i) $S \rightarrow SdB'b \mid cB'b \mid a$

$B' \rightarrow SdB' \mid eB'$

$B' \rightarrow cB' \mid \varepsilon$

(ii) After removing immediate left recursion is

$S \rightarrow SdB'b$

$S \rightarrow eB'bS' \mid aS'$

$S' \rightarrow dB'bS' \mid \varepsilon$

$B \rightarrow SdB' \mid eB$

$B' \rightarrow cB' \mid \varepsilon$

The grammar is now free from left recursion.

b) What is Operator Precedence Parsing? Discuss about the advantage and disadvantage of Operator Precedence Parsing. [WBUT 2012, 2015]

Consider the following grammar:

$E \rightarrow TA$

$A \rightarrow +TA \mid \varepsilon$

$T \rightarrow FB$

$B \rightarrow *FB \mid \varepsilon$

$F \rightarrow id$

Test whether this grammar is Operator Precedence Grammar or not and show how the string $w = id + id * id + id$ will be processed by this grammar. [WBUT 2012]

Answer:

1st Part:

An operator-precedence parser is a simple shift-reduce parser capable of parsing a subset of LR (1) grammars. More precisely, the operator precedence parser can parse all LR (1) grammars where two consecutive non-terminals never appear in the right-hand side of any rule.

The technique of parsing through this parser is Operator Precedence Parsing.

2nd Part:

Advantage:

- Because of its simplicity numerous compiler using operator precedence parsing technique.
- Can have been built for the entire language.

Disadvantage:

- Hard to handle token like unary minus
- Worse, since relation between a grammar for the language being parsed and the operator precedence parser itself is week. That is cannot always be sure the parser accepts exactly the desired language.

Last Part:

The given grammar is NOT an operator precedence grammar. It is not even an operator grammar because:

(i) There are several productions (e.g., $T \rightarrow FB$) where two non-terminals occur side by side in the right hand side.

(ii) There are several ε productions.

The string $w = id + id * id + id$ CANNOT be parsed by the given grammar since the terminal '=' is not in the terminal-set of the grammar.

2. Write short note on Left Recursion. **[WBUT 2018]**

Answer:

Refer to Question No. 1(a) of Long Answer Type Questions.

TOP-DOWN PARSING

Multiple Choice Type Questions

1. A dangling reference is a [WBUT 2006, 2008]
   a) pointer pointing to storage which is freed
   b) pointer pointing to nothing
   c) pointer pointing to storage which is still in use
   d) pointer pointing to uninitialized storage

Answer: (a)

2. Which of the following is not a loop optimization? [WBUT 2008, 2009]
   a) Loop unrolling
   b) Loop jamming
   c) Loop heading
   d) Induction variable elimination

Answer: (c)

3. A top down parser generates [WBUT 2010, 2012, 2013, 2015]
   a) leftmost – derivation
   b) rightmost – derivation
   c) leftmost derivation in reverse
   d) rightmost derivation in reverse

Answer: (a)

4. $FIRST(\alpha\beta)$ is [WBUT 2011]
   a) $FIRST(\alpha)$
   b) $FIRST(\alpha) \cup FIRST(\beta)$
   c) $FIRST(\alpha) \cup FIRST(\beta)$ *if* $FIRST(\alpha)$ contains ε else $FIRST(\alpha)$
   d) none of these

Answer: (c)

5. A given grammar is not LL(1) if the parsing table of a grammar may contain
   a) any blank filed
   b) any e-entry [WBUT 2011, 2019]
   c) duplicate entry of same production
   d) more than one production rule

Answer: (d)

6. First pos of a (dot) node with leaves c1 and c2 is [WBUT 2011]
   a) $firstpos(c1) \cup firstpos(c2)$
   b) $firstpos(c1) \cap firstpos(c2)$
   c) $if\,(nullable(c1))\,firstpos(c1) \cup firstpos(c2)\,else\,firstpos(c1)$
   d) $if\,(nullable(c2))\,firstpos(c1) \cup firstpos(c2)\,else\,firstpos(c1)$

Answer: (c)

7. Left factoring guarantees [WBUT 2011, 2018]
   a) not occurring of backtracking
   b) cycle free parse tree
   c) error free target code
   d) correct LL(1) parsing table

Answer: (a)

8. Given the grammar S → ABc, A → a | ε, B → b | ε, FOLLOW(A) is the set [WBUT 2015]

a) {$} b) {b} c) {b, c} d) {a, b, c}

Answer: (c)

9. Which one of the following is a top-down parser? [WBUT 2017]

a) Recursive descent parser b) Operator precedence parser

c) An LR(k) parser d) An LALR(k) parser

Answer: (a)

Short Answer Type Questions

1. Eliminate left recursion from the following grammar: [WBUT 2007, 2014]

E →E +T|T

T →TF|F

F →F * |a|b

Answer:

After eliminating the left recursion, the resulting grammar (with three additional non-terminals E' and T') have the rules:

$$E \to TE'$$

$$E' \to +TE' \mid \varepsilon$$

$$T \to FT'$$

$$T' \to FT' \mid \varepsilon$$

$$F \to aF' \mid bF'$$

$$F' \to *F' \mid \varepsilon$$

2. What is recursive descent parsing? Describe the drawbacks of recursive descent parsing for generating the string abc from the grammar: [WBUT 2011]

S→ aBx

B → bc | b

Answer:

Recursive descent parsing is a simple way to construct a top-down parser by regarding each production rule as a function, where:

- The name of the function is the non-terminal on the left hand side of the rule
- Each instance of a non-terminal on the right hand side is a call to the corresponding function.
- Each instance of a terminal on the right hand side tells us to match it with the input symbol and thereby 'consume' it.
- Parsing happens as the execution of the function corresponding to the start symbol.

In the given grammar, we have problem is constructing the function for the non-terminal B because both the B productions start with the terminal b.

3. What is recursive descent parsing? Describe the drawbacks of recursive descent parsing for generating the string 'abc' from the grammar:

$S \rightarrow aBc$

$B \rightarrow bc \mid b$

[WBUT 2017, 2018]

Answer:

1st Part: ***Refer to Question No. 2 (1st Part) of Short Answer Type Questions.***

2nd Part:

Recursive Decent Parsing:

$S \rightarrow aBc$
$B \rightarrow bc \mid b$
} drawbacks of generating *abc* from the given grammar

⇒ The recursive decent parser uses the concept of backtracking while parsing.

Features	Top-down Approach
1. Technique used	This technique uses left most derivation.
2. Level strategy	It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar.
3. Definition	Top-down parsing attempts to find the left most derivations for an input string.
4. Goal	In this approach the main decision is to select what production rule to use in order to construct the string.

String → Grammar → R D Parser → Parse Tree

Now, here we have the grammar—

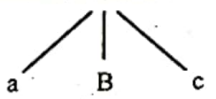
$S \rightarrow aBc$

$B \rightarrow bc \mid b$

String w : abc

i/p ptr

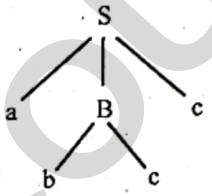
Step 1:



Decent ptr

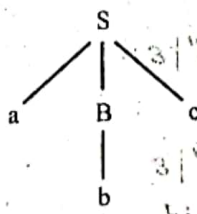
Now, i/p and decent ptr matched, so we go on increasing both the pointer.

Step 2:



Decent ptr

OR



Decent ptr

Now, for the recursive decent parser, it will give chance to each and every production, if not a matched production, then it will backtracked again use the other optional production.

For example: Here $B \to bc$ or $B \to b$

Now, the parser can't determine which production is useful to generate abc.

Now, it is a drawback of this parser as, every time if the wrong production is chosen then the parser needs to backtrack and again use other available productions.

So, to solve this drawback of recursive decent parser, the concept of predictive parsing is used frequently.

4. Differentiate between top-down and bottom-up approach. [WBUT 2019]

Answer:

Features	Top-down Approach	Bottom-up approach
1.Technique used	This technique uses left most derivation.	This technique uses right most derivation.
2. Level strategy	It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar.	It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar.
3. Definition	Top-down parsing attempts to find the left most derivations for an input string.	Bottom-up parsing can be defined as an attempt to reduce the input string to start symbol of a grammar.
4. Goal	In this approach the main decision is to select what production rule to use in order to construct the string.	In this approach the main decision is to select when to use a production rule to reduce the string to get the starting symbol.

Long Answer Type Questions

1. Which parser is used for the implementation of recursive descent parsing? Draw a model diagram for that parser. Construct the parsing table for the grammar:

$$E \to TE'$$
$$E' \to +TE' \mid \varepsilon$$
$$T \to FT'$$
$$T' \to *FT' \mid \varepsilon$$
$$F \to (E) \mid id$$

With the help of parsing table and other components how the string id + id * id is parsed?

[WBUT 2006]

OR,

Consider the following grammar:

$E \rightarrow E + T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

(i) Obtain the FIRST and FOLLOW sets for the above grammar.

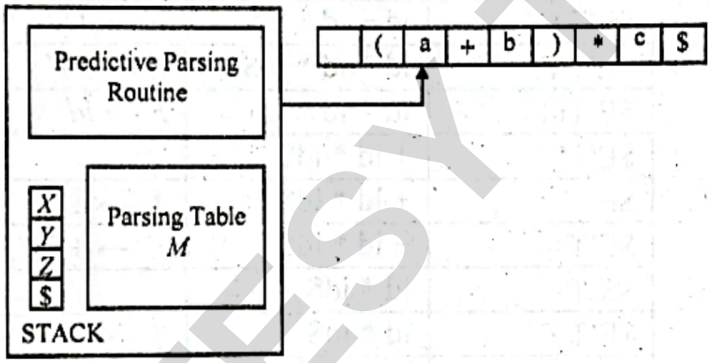
(ii) Construct the predictive parsing table for the above grammar. **[WBUT 2015]**

Answer:

1st Part:

A commonly used non-recursive (in program code) implementation of a recursive descent parser is the Predictive Parser.

If we recall our lessons in Formal Language Theory, we may perhaps remember that the automata for a CFG is a Push-Down Automata (PDA). It may be worthwhile mentioning here that an LL parser is in effect an implementation of PDA. Now, a PDA was defined as working with a "push-down store" or stack. Where is this stack in the recursive descent parser? The stack in recursive descent parser is the same stack that programs use while using subroutine calls. As we have seen, a trivial implementation of a recursive parser for a given grammar (without left-recursion, say) may require too many backtracking. However, the backtracking may be eliminated by using the concepts of left-recursion removal, FIRST, FOLLOW, etc.



Predictive Parser in Action

2nd Part:

Clearly, **FIRST(F)** = {(,id}.

Also, **FIRST(T′)** = {*,ε} and **FIRST(E′)** = {+,ε}.

Now **FIRST(E)** contains everything in **FIRST(T)** and **FIRST(T)** contains everything in **FIRST(F)**.

From this we get:

FIRST(E) = **FIRST(T)** = **FIRST(F)** = {(,id}.

Next, we can start with **FOLLOW(E)** = {),$}.

FOLLOW(T) = **FIRST(E′)** − {ε} = {+}.

FOLLOW(E′) = **FOLLOW(E)** = {),$}.

FOLLOW(T′) = **FOLLOW(T)** = {+} because **FOLLOW(T)** contains all non-ε symbols in **FIRST(E′)**

Also, since ε ∈ **FIRST(E′)**,

we get **FOLLOW(T) = FOLLOW(T′) = {+,),$}.**
FOLLOW(F) = FIRST(T′) – {ε} = {*} because **FOLLOW(F)** contains all non-ε symbols in **FIRST(T′).**
Also, since ε ∈ **FIRST(T′)**, we get
FOLLOW(F) = {+,*,),$}.

The Predictive Parsing Table is:

Non-Terminal	INPUT SYMBOL					
	+	*	(	)	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 → (E)		F → id	

The Predictive Parser's action for input id + id *id$ is as shown below:

Stack	Input	Output
$E	id + id *id$	
$E′T	id + id * id$	$E \rightarrow TE'$
$E′T′F	id + id * id$	$E \rightarrow FT'$
$E′Tid	id + id * id$	$F \rightarrow id$
$E′T′	+ id * id$	
$E′	+ id * id$	$T' \rightarrow \varepsilon$
$E′T+	+ id * id$	$E' \rightarrow +TE'$
$E′T	id * id$	
$E′T′F	id * id$	$T \rightarrow FT'$
$E′T′id	id * id$	$F \rightarrow id$
$E′T′	* id$	
$E′T′F*	* id$	$T' \rightarrow *FT'$
$E′T′F	id$	
$E′T′id	id$	$E \rightarrow id$
$E′T′	$	
$E′	$	$T' \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$

2. Design an LL(1) parsing table for the following grammar: **[WBUT 2008]**
S→aAcd|BcAe
A→b|ε
B→C f |d
C→f e
With the help of the parsing table, show how the string fefcbe is parsed.

Answer:
Clearly, the given grammar is free from any left-recursion — there is not even any immediate left-recursion. So, we begin by computing the FIRST and FOLLOW sets for the non-terminals:

FIRST(S) = {a,d, f }, FIRST(A) = {b, ε}, FIRST(B) = {d, f } and FIRST(C) = {f }.
FOLLOW(S) = {$}, FOLLOW(A) = {c,e}, FOLLOW(B) = {}, and FOLLOW(C) = {f }.
The parsing table is therefore:

	a	b	c	d	e	f	$
S	$S \rightarrow aAcd$			$S \rightarrow BcAe$		$S \rightarrow BcAe$	
A		$A \rightarrow b$	$A \rightarrow \varepsilon$		$A \rightarrow \varepsilon$		
B				$B \rightarrow d$		$B \rightarrow Cf$	
C						$C \rightarrow fe$	

The parse for the string fefcbe proceeds as follows:

S → aAcd|BcAe
A→ b|ε
B → C f |d
C → f e

Stack	Input	Output/Action
$S	fefcbe$	START
$eAcB	fefcbe$	S → BcAe
$eAcfC	fefcbe$	B → Cf
$eAcfef	fefcbe$	C → fe
$eAcfe	efcbe$	POP
$eAcf	fcbe$	POP
$eAc	cbe$	POP
$eA	be$	POP
$eb	be$	A → b
$e	e$	POP
$	$	SUCCESS

3. a) Consider the following Grammar: **[WBUT 2008]**

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

i) Obtain the FIRST and FOLLOW sets for the above grammar.
ii) Construct the Predictive Parsing table for the above grammar.
b) Explain the Predictive Parser's action by describing the moves it would make on an input id + id *id$.

Answer: ***Refer to Question No. 1 of Long Answer Type Questions.***

4. a) Describe with a block diagram the parsing technique of LL(1) parser. **[WBUT 2011, 2018]**

b) Parse the string abba using LL(1) parser where the parsing table is given below: **[WBUT 2011, 2018]**

	A	b	$
S	S → aBa		
B	B → e	B → bB	

c) Check whether the following grammar is LL(1) or not. **[WBUT 2011]**

$S \rightarrow iCtSE \mid a$

$E \rightarrow eS \mid \varepsilon$

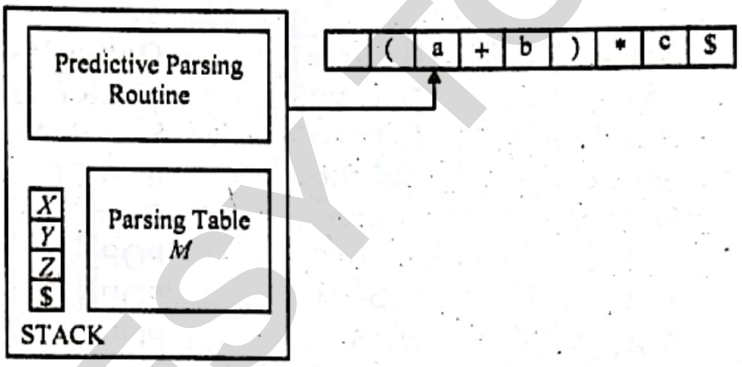
$C \rightarrow b$

Answer:

a) LL(1) parsers can be of various types. However from the question it appears that a description of a "Non-recursive Predictive Parser" is to be given.

A Non-Recursive Predictive Parser is an LL parser implementation based on a table driven algorithm that uses an explicit stack.

The block diagram of a typical predictive parser is shown below:



Predictive Parser in Action

The parsing table is of type $M[A,a]$ which indicates which production to use if the top of the stack is a non-terminal A and the current token is equal to a. If there is a valid (i.e., nor "error") entry $M[A,a]$, we pop A from the stack and push all the right-hand side symbols of the production stored in the entry $M[A,a]$. If that happens to be $A \rightarrow XYZ$, we push symbols into the stack in the reverse order, i.e., we push Z first, followed by Y and X. We use

a special symbol $ to denote the end of file.

Assuming S to be the start symbol, the non-recursive predictive parsing algorithm is as follows:

```
push(S);
a = CurrentInput();
do {
X = pop();
if (X is a terminal or '$') {
```

```
if (X == a) {
ConsumeInput();
a = CurrentInput();
}
else Error();
else if (M[X,a] == "X ? Y1 Y2 ... Yk") {
push(Yk);
...
push(Y1);
}
else Error();
} while X != '$';
```

b) The parsing action is summarized in the table below:

Stack	Input	Output
$S	abba$	
$aBa	abba$	s → aBa
$aB	bba$	
$aBb	bba$	B → bB
$aB	ba$	
$aBb	ba$	B → bB
$aB	a$	
$a	a$	B → e
$	$	

c) From grammar rules and properties of FIRST and FOLLOW we get:

FIRST(S) = {i,a}.

FIRST(E) = {e,ε}.FIRST(C)={b}.

FOLLOW(S) = {$}$\cup$FIRST(E) = {e,$}.

FOLLOW(E) = {e,$}.

FOLLOW(C) = {t}.

The predictive parsing table then becomes:

Non-terminal	Input Symbol					
	I	t	e	a	b	$
S	S→iCtSE			S → a		
E			E→eS E → e			S → e
C					C→ b	

Clearly, there is a duplicate entry in $P[E,e]$ contains multiple entries and hence the grammar is not LL(1).

5. Consider the following grammar: **[WBUT 2013]**

S → aABb
A → c | ε
B → d | ε

Prove the above grammar is LL (1).
Draw the parsing table.
Now check whether the string "ab" and "acdb" are the languages of the above grammar.
(Derive each step with the help of a stack.)

Answer:

S → aABb
A → c | ε
B → d | ε

FIRST(S) = FIRST(aABb) = { a }
FIRST(A) = FIRST(c) U FIRST(ε) = { c, ε }
FIRST(B) = FIRST(d) U FIRST(ε) = { d, ε }
Since the right-end marker $ is used to mark the bottom of the stack, $ will initially be immediately below S (the start symbol) on the stack; and hence, $ will be in the FOLLOW(S). Therefore:
FOLLOW(S) = {$}
Using S → aABb, we get:
FOLLOW(A) = FIRST(Bb)
= FIRST(B) – {∈} ∪ FIRST(b)
= {d, ∈} – {∈} ∪ {b} = {d, b}
FOLLOW(B) = FIRST(b) = {b}
Therefore, the parsing table is as shown in Table below.

Table : Production Selections for Parsing Derivations

	a	*b*	*c*	*d*	$
S	*S → aABb*				
A		*A → ∈*	*A → c*	*A → ∈*	
B		*B → ∈*		*B → d*	

As there is only single production in each entry so the grammar is LL(1).

Consider an input string *acdb*. The various steps in the parsing of this string, in terms of the contents of the stack and unspent input, are shown in Table below.

Stack Contents	Unspent Input	Moves
$S	acdb$	Derivation using S → aABb
$bBAa	acdb$	Popping a off the stack and advancing one position in the input
$bBA	cdb$	Derivation using A → c
$bBc	cdb$	Popping c off the stack and advancing one position in the input
$bB	db$	Derivation using B → d
$bd	db$	Popping d off the stack and advancing one position in the input
$b	b$	Popping b off the stack and advancing one position in the input
$	$	Announce successful completion of the parsing

Similarly, for the input string "ab", the various steps in the parsing of the string, in terms of the contents of the stack and unspent input, are shown in Table below.

Stack Contents	Unspent Input	Moves
$S	ab$	Derivation using S → aABb
$bBAa	ab$	Popping a off the stack and advancing one position in the input
$bBA	b$	Derivation using A →ε
$bB	b$	Derivation using B → ε
$b	b$	Popping b off the stack and advancing one position in the input
$	$	Announce successful completion of the parsing

6. a) Consider the following grammar and design a SLR parser table: [WBUT 2016]

S → AA
A → aA
A → b

b) Make a comparison between Predictive Parser and Shift Reduce Parser.

Answer:

a)

0: $S' \to S\$$

1: $S \to AA$

2: $A \to aA$

3: $A \to b$

	First	Follow
S	(a,b)	$\$$
A	(a,b)	$(a,b,\$)$

$I_0 - (S' \to .S\$,\ S \to .AA,\ A \to .aA,\ A \to .b)$

$(I_0, S) \to I_1 \to (S' \to S.)$

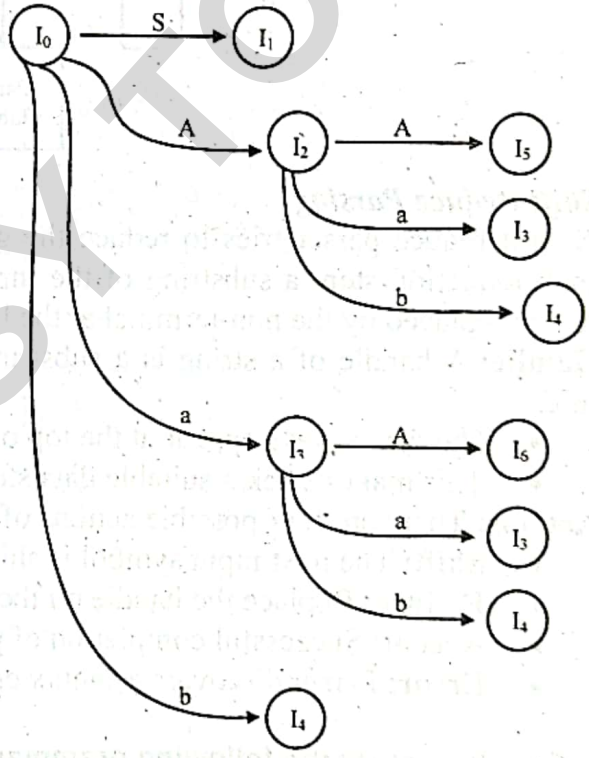
$(I_0, A) \to I_2 \to (S \to A.A, A \to .aA,\ A \to .b)$

$(I_0, a) \to I_3 \to (A \to a.A, A \to .aA,\ A \to .b)$

$(I_0, b) \to I_4 \to (A \to b.)$

$(I_2, A) \to I_5 \to (S \to AA.)$

$(I_3, A) \to I_6 \to (A \to aA.)$



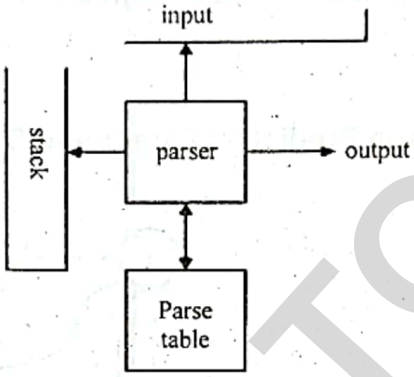
State	action			goto		
	a	b	$	S'	S	A
I_0	S_3	S_4			1	2
I_1				acc		
I_2	S_3	S_4				5
I_3	S_3	S_4				6
I_4	r_3	r_3	r_3			
I_5			r_1			
I_6	r_2	r_2	r_2			

b) *Predictive Parser*

- A non recursive top down parsing method
- Parser "predicts" which production to use
- It removes backtracking by fixing one production for every non-terminal and input token(s)
- Predictive parsers accept LL(k) languages

First L stands for left to right scan of input
Second L stands for leftmost derivation
k stands for number of lookahead token

- In practice LL(1) is used



Shift Reduce Parsing:

A shift reduce parser tries to reduce the given input string into the starting symbol. At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.

Handle: A handle of a string is a substring that matches the right side of a production rule.

- Handles always appear at the top of the stack and never inside it.
- This makes stack a suitable data structure.

Actions: There are four possible actions of a shift reduce parser

- **Shift:** The next input symbol is shifted onto the top of the stack.
- **Reduce:** Replace the handle on the top of the stack by the non-terminal.
- **Accept:** Successful completion of parsing.
- **Error:** Parser discovers a syntax error and calls an error recovery routine.

7. Check whether the following grammar is LL(1) or not: **[WBUT 2018]**

$X \rightarrow Yz \mid a$

$Y \rightarrow bZ \mid \varepsilon$

$Z \rightarrow \varepsilon$

Answer: ***Similar to Question No. 4(c) of Long Answer Type Questions.***

BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY

Multiple Choice Type Questions

1. YACC builds up [WBUT 2006, 2007, 2010, 2011, 2014, 2017, 2018, 2019]
   a) SLR parsing table
   b) canonical LR parsing table
   c) LALR parsing table
   d) none of these

Answer: (c)

2. If a grammar is LALR (1) then it is necessarily: [WBUT 2006, 2007, 2008, 2010, 2016]
   a) SLR(1) b) LR(1) c) LL(1) d) none of these

Answer: (b)

3. Which data structure is mainly used during shift-reduce parsing? [WBUT 2007, 2008, 2014, 2016]
   a) Pointers b) Arrays c) Stacks d) Queues

Answer: (c)

4. If I is a set of valid items for a viable prefix γ, then GOTO (I , X) is a set of items that are valid for the viable prefix: [WBUT 2009]
   a) δ b) γ c) Prefix of γ d) None of these

Answer: (a)

5. Shift-reduce parsers are [WBUT 2009, 2014, 2016, 2018]
   a) Top-down parsers
   b) Bottom-up parsers
   c) May be top-down or bottom-up parsers
   d) None of these

Answer: (b)

6. Which of the following is the most powerful parser? [WBUT 2013]
   a) SLR
   b) LALR
   c) Canonical LR
   d) Operator precedence

Answer: (b)

7. A parser with the valid prefix property is advantageous because it [WBUT 2013]
   a) detect errors as soon as possible
   b) detect errors as and when they occur
   c) limits the amount of erroneous output passed to the next phases
   d) all of these phases

Answer: (c)

8. Which of the following is an example of bottom up parsing? [WBUT 2017]

a) LL parsing
b) Predictive Parsing
c) Recursive descent parsing
d) Shift-reduce parsing

Answer: (d)

9. Synthesized attribute can easily be simulated by an [WBUT 2017]

a) LL grammar
b) LR grammar
c) ambiguous grammar
d) none of these

Answer: (b)

Short Answer Type Questions

1. Explain with an example. What is the reduce-reduce conflict in LR parser? [WBUT 2013]

Answer:

For any two complete items $A \rightarrow a.$ and $B \rightarrow b.$ in s, Follow(A) and Follow(B) are disjoint (their intersection is the empty set). Violation of this rule is a Reduce-Reduce Conflict.

2. Construct SLR parsing table for the grammar given below. [WBUT 2018]

$S \rightarrow Ab$
$A \rightarrow bA/a$

Answer: *Similar to Question No. 4(c) of Long Answer Type Questions.*

Long Answer Type Questions

1. Given a grammar G=({E,T,F},{id,+,*, (,)},P,E) , where, P is given by:
E →E +T|T,
T →T *F|F,
F→(E)|id
Construct the SLR(1) parsing table for G. [WBUT 2006, 2009]

Answer:

The augmented grammar is:

$E' \rightarrow E$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

Now, $s_0 = \text{closure}(E' \rightarrow .E.)$

We start with $s_0 = \{\mathbf{E}' \to .\mathbf{E}\}$. Silence the 'dot' is in front of **E**, a non-terminal, items for all **E** productions with the dot at the beginning, will be appended with the s_0. So, s_0 becomes:

$s_0 = \{\mathbf{E}' \to .\mathbf{E}, \mathbf{E} \to .\mathbf{E}+\mathbf{T}, \mathbf{E} \to .\mathbf{T}\}$.

Again, the dot is front of a (new) non-terminal T and all T productions with the dot at the beginning, will be appended to s_0. So, s_0 becomes:

$s_0 = \{\mathbf{E}' \to .\mathbf{E}, \mathbf{E} \to .\mathbf{E}+\mathbf{T},\}$.

$E \to .T, T \to .T * F, T \to .F.$

Again, the dot is front of a (new) non-terminal F and all F productions with the dot at the beginning, will be appended to s_0. So, finally, s_0 becomes:

$s_0 = \{\mathbf{E}' \to .\mathbf{E}, \mathbf{E} \to .\mathbf{E}+\mathbf{T},.\}$

$E \to .T, T \to .T * F, T \to .F, F \to .(E), F \to .id$

It is possible to have transitions from s_0 on **E, T, F, id** and (,because, these are the grammar symbols in the items in s_0, where the dot is just before it.

Consider:

$s_1 = \text{goto}(s_0, \mathbf{E})$.

This is simple. Simply let the dot cross **E** wherever the dot is in front of **E**.

This gives:

$s_1 = \{\mathbf{E}' \to \mathbf{E}., \mathbf{E} \to \mathbf{E}.+\mathbf{T}\}$.

Note that in both the items, the dot is in front of terminals. So, the "closure" operation will not add further items.

Similarly,

$s_2 = goto(s_0, \mathbf{T}) = \{\mathbf{E} \to \mathbf{T}., \mathbf{T} \to \mathbf{T}.*\mathbf{F}\}$

$s_3 = goto(s_0, \mathbf{F}) = \{\mathbf{T} \to \mathbf{F}.\}$

$s_5 = goto(s_0, \mathbf{id}) = \{\mathbf{E} \to \mathbf{id}.\}$

Much more interesting is $s_4 = goto(s_0, (\ )$

This start with:

$s_4 = \{\mathbf{F} \to (.\mathbf{E})\}$

and then with repeated application of closure, end with:

$s_4 = \{\mathbf{F} \to (.\mathbf{E}), \mathbf{E} \to .\mathbf{E} + \mathbf{T}, \mathbf{E} \to .\mathbf{T},\}$

$T \to .T * F, T \to .F, F \to .(E), F \to .id.$

We can proceed similarly to get the following:

$s_6 = goto(s_1, +) = \{\mathbf{E} \to \mathbf{E} + .\mathbf{T}, \mathbf{T} \to \mathbf{T}*\mathbf{F},\}$

$T \to .F, F \to .(E), F \to .id.$

$s_7 = goto(s_2, *) = \{T \to T*.F, F \to .(F), F \to .id\}.$

$s_8 = goto(s_4, E) = \{F \to (E.), E \to E.+T\}.$

$s_9 = goto(s_6, T) = \{E \to E+T., T \to T.*F\}.$

$s_{10} = goto(s_7, F) = \{T \to T*F.\}.$

$s_{11} = goto(s_8,)) = \{F \to (E).\}.$

Additionally:

$goto(s_4, T) = s_2$

$goto(s_4, F) = s_3.$

$goto(s_4, () = s_4.$

$goto(s_4, id) = s_5.$

$goto(s_6, F) = s_3.$

$goto(s_6, () = s_4.$

$goto(s_6, id) = s_5.$

$goto(s_7,) = s_4.$

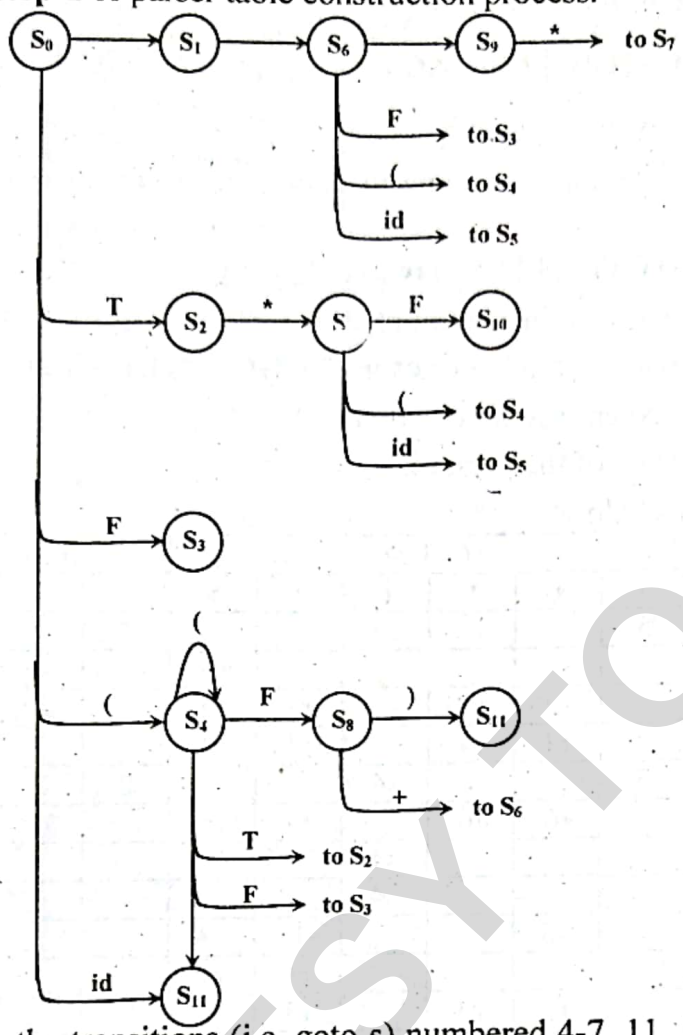
$goto(s_7, id) = s_5.$

$goto(s_8, +) = s_6.$

$goto(s_9, *) = s_7.$

You can clearly visualize the DFA and are encouraged to draw a diagram for the same (see Fig).

We have completed **Step-1** of parser table construction process.



In **Step-2,** we can use the transitions (i.e. goto-s) numbered 4-7, 11, 14, 15 and 17-22 to create the **shift-j** entires of the **ACTION** part of the parsing table. For example:

- Using goto $(s_4, ( ) = s_4$, we get **ACTION** $[4', ('] = s^4$.
- Similarly, from goto $(s^4, \mathbf{id}) = s_5$ and goto $(s^6, \mathbf{id}) = s_5$, we get **ACTION** $[6', \mathbf{id}'] = s^5$. as also **ACTION** $[7', \mathbf{id}'] = s^5$.
- Again, since goto $(s_8,)) = s_{11}$, we get **ACTION** $[8',)'] = s^{11}$.

Use the other transitions to complete the **shift** entires of the parsing table. For **Step-3,** we need to calculate the **FOLLOW** sets for non terminals **E, T** and **F**. We have already seen how to calculate the **FOLLOW** sets. So we know **FOLLOW (E)** = $\{\$, +,)\}$. Now consider the item $\mathbf{E} \to \mathbf{T}$. in s_2. The item comes from rule-2. Because of this, the entires **ACTION**$[2, \$]$, **ACTION** $[2, +]$ and **ACTION** $[2,)]$ are all **reduce by** $\mathbf{E} \to \mathbf{T}$ or r_2.

By similar logic, since **FOLLOW (F)** $=\{+,*,),\$\}$ and $\mathbf{F \rightarrow id}$. (which comes from rule-6) is in s_5. we get **ACTION** $[2,\$]$, **ACTION** $[2,+]$, **ACTION** $[2,*)]$ and **ACTION** $[2,\ )]$ are all **reduce by** $\mathbf{F \rightarrow id}$ or r_6.

We can use the other appropriate items to complete the **reduce** entires of the parsing table.

Using **Step-4,** we get **ACTION** $[1,\$]$**=acc.**

Using **Step-5** and the transitions numbered 1-3, 8-10, 12, 13 and 16, we get all the **GOTO** entires of the parsing table. For example, **GOTO[4,T]=2**, since $goto(s_4, \mathbf{T}) = s_2$.

By **Step-6,** All non-filled entires are error entires.

By **Step-7,** the initial state of the parser is s_0.

The Parsing Table is as below:

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

2. What is shift reduce parsing? What is handle? Give examples of handle. Show an illustration of shift reduce parsing for a suitable grammar and for Each reduction indicate the corresponding handle. **[WBUT 2007]**

OR,

What is a handle? Describe various actions of a shift reduce parser. **[WBUT 2014]**

Answer:

Shift-reduce parsing is a general style of bottom-up syntax analysis. This technique attempts to construct a parse tree for the input string beginning at the leaves and working up to the root. One can think of this as one of "reducing" a string w to the start symbol. At each reduction step, a particular substring the right side of a production is replaced by the symbol on the left of that production and if the substring chosen is correct at each step, a rightmost derivation is traced out in reverse.

A handle of a right sentential form gamma is a production A →b and a position in gamma where the string beta may be found and replaced by A to get the previous right-sentential form.

Consider the grammar:
E ⇒E +E|E *E|a|b|c
One rightmost derivation for a+b*c is (handles underlined):
E ⇒ E +E ⇒ E +E *E ⇒ E +E * c ⇒ E +b * c ⇒ a+b * c
The reductions that happen in order are obtained from the last derivation by replacing the underlined handle by the non-terminal on the left (in this case, E).

3. Construct SLR parsing table for following grammar: **[WBUT 2009]**

S → AS | b
A → SA | a

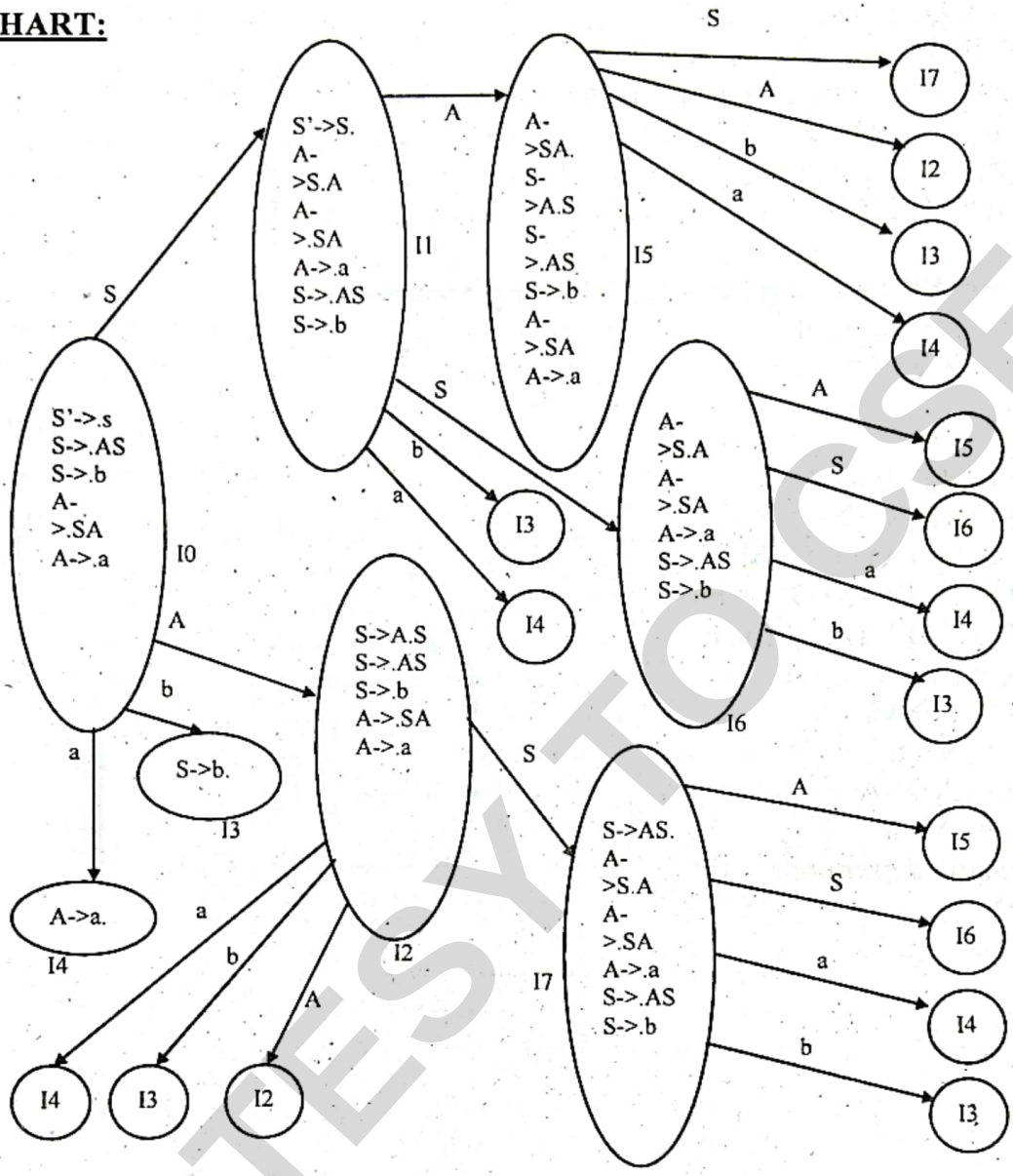
Answer:

Part A

- ***First and Follow Set:***
  FIRST (S) → {a, b}
  FIRST (A) → {a, b}
  FOLLOW (S) → {a, b, $}
  FOLLOW (A) → {a, b}
- ***Reductions:***
  r1= S→AS
  r2= S→b
  r3= A→SA
  r4= A→a
- ***Augmented grammar:***

S'→S
S→AS
S→b
A→SA
A→a

➢ **CHART:**



SLR parsing table:

STATE	ACTION			GOTO	
	A	b	$	S	A
0	s4	s3	acc	1	2
1	s4	s3		6	5
2	s4	s3		7	2
3	r2	r2	r2		
4	r4	r4			
5	s4	s3		7	2
6	s4	s3		6	5
7	s4	s3		6	5

The grammar is in SLR as because there is no conflict.

4. a) Describe LR parsing with block diagram. **[WBUT 2011, 2014, 2018]**

b) What are the main advantages of LR parsing? **[WBUT 2011, 2018]**

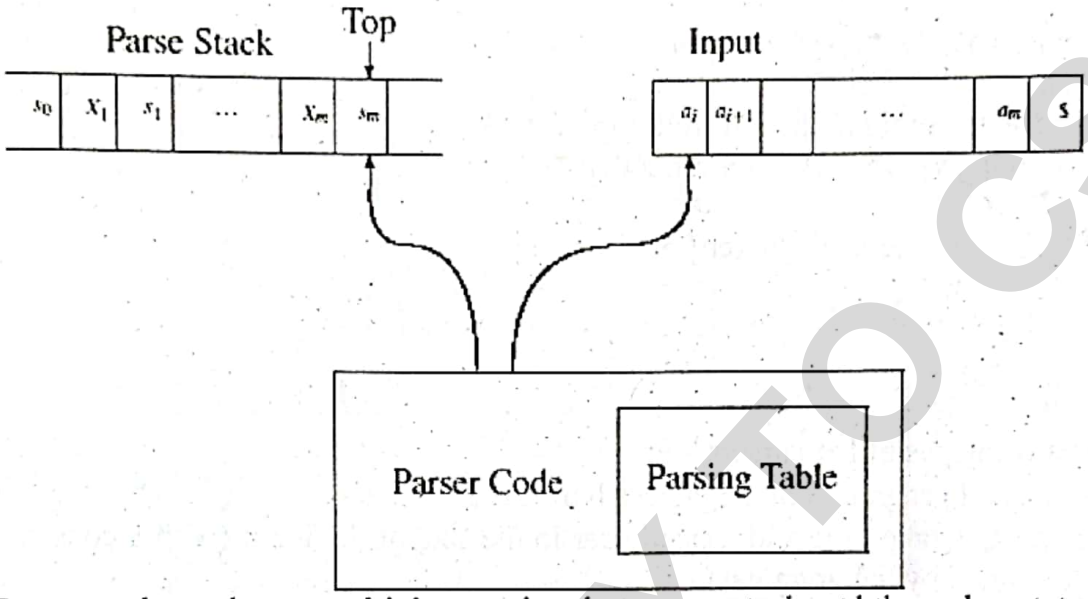
c) Construct SLR parsing table for the grammar given below:

$S \rightarrow Cb$

$C \rightarrow bC \mid d$ **[WBUT 2011]**

Answer:

a) The block diagram of an LR parser is given below:



All LR-parsers have the same driving routine that uses a stack and through a state-based approach, tries to detect whether the stack contains the handle at its top. If so, the handle is replaced by the left hand side non-terminal for the handle.

The parsing algorithm uses two tables — ACTION and GOTO. The behavior can be summarized again as follows.

The parser sees s_m at the top of the stack and let a_i be the current input symbol.

Based on these, the parser can carry out one of the following operations:

- Shift-s. Here s is a state.
- Reduce by A → b
- Accept
- Error

Assuming that states are numbered $s_0, s_1, \ldots$ where s0 is a special case such that the parser begins with only s0 on the stack, the driving routine of an LR parser is:

```
push(0);
read_next_token();
for(;;)
{
s = top(); /* current state is taken from top of stack */
if (ACTION[s,current_token] == "s-i")
/* shift and go to state i */
```

```
{
push(current_token);
push(i);
read_next_token();
}
else if (ACTION[s,current_token] == "r-i")
/* reduce by rule i: X ? A1...An */
{
perform pop() 2 * n times;
s = top();
/* restore state before reduction from top of stack */
push(GOTO[s,X]); /* state after reduction */
}
else if (ACTION[s,current_token] == "succ")
success!!
else error();
}
```

b) The advantages of LR parsers are:

- Almost all programming languages have LR grammars.
- LR parsers take time and space linear in the size of the input (with a constant factor determined by the grammar).
- LR is strictly more powerful than LL (for example, every LL(1) grammar is also both LALR(1) and LR(1), but not vice versa).
- LR grammars are more "natural" than LL grammars (e.g., the grammars for expression languages get mangled when we remove the left recursion to make them LL(1), but that is not necessary for an LR(1) grammar).

c) We construct the augmented grammar:

$0. S' \rightarrow S$

$1. S \rightarrow Cb$

$2. C \rightarrow bC$

$3. C \rightarrow d$

Also, we note that

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

$FOLLOW(C) = \{b\}$

We proceed to make the canonical collection of LR(0) items as follows:

$s_0 = CLOSURE(\{S' \rightarrow .S\}) = \{S' \rightarrow .S, S \rightarrow .Cb, C \rightarrow .bC, C \rightarrow .d\}$

$s_1 = CLOSURE(GOTO[s_0, S]) = \{S' \rightarrow S.\}$

$s_2 = CLOSURE(GOTO[s_0, C]) = \{S \rightarrow C.b\}$

$s_3 = CLOSURE(GOTO[s_0, b]) = \{C \rightarrow b.C, C \rightarrow .bC, C \rightarrow .d\}$

$s_4 = CLOSURE(GOTO[s_0, d]) = \{C \to d.\}$

$s_5 = CLOSURE(GOTO[s_2, b]) = \{S \to Cb.\}$

$s_6 = CLOSURE(GOTO[s_3, C]) = \{S \to bC\}$

$GOTO(s_3, b) = s_3$

$GOTO(s_3, d) = s_4$

SLR parsing table:

STATE	ACTION			GOTO	
	A	b	$	S	A
0	s4	s3	acc	1	2
1	s4	s3		6	5
2	s4	s3		7	2
3	r2	r2	r2		
4	r4	r4			
5	s4	s3		7	2
6	s4	s3		6	5
7	s4	s3		6	5

The grammar is in SLR as because there is no conflict.

5. Consider the following grammar: **[WBUT 2013]**

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to id$

Draw a SLR state transition diagram for the above grammar. Also draw the SLR parse table.

Answer:

The augmented grammar is:

$E' \to E$

$E \to E + T$

$E \to T$

$T \to T * F$

$T \to F$

$F \to id$

Now, $s_0 = \text{closure}(E' \to .E)$

We start with $s_0 = \{E' \to .E\}$. Silence the 'dot' is in front of E, a non-terminal, items for all E productions with the dot at the beginning, will be appended with the s_0. So, s_0 becomes:

$s_0 = \{E' \to .E, E \to .E+T, E \to .T\}$.

Again, the dot is front of a (new) non-terminal T and all T productions with the dot at the beginning, will be appended to s_0. So, s_0 becomes:

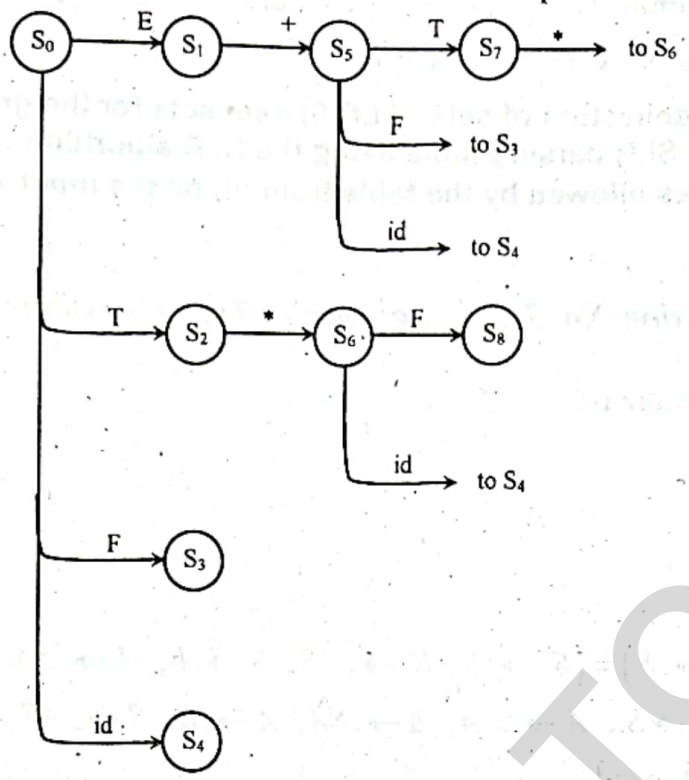
$s_0 = \{\mathbf{E}' \rightarrow .\mathbf{E}, \mathbf{E} \rightarrow .\mathbf{E}+\mathbf{T},\}.$
$E \rightarrow .T, T \rightarrow .T * F, T \rightarrow .F.$
Again, the dot is front of a (new) non-terminal F and all F productions with the dot at the beginning, will be appended to s_0. So, finally, s_0 becomes:

$s_0 = \{\mathbf{E}' \rightarrow .\mathbf{E}, \mathbf{E} \rightarrow .\mathbf{E}+\mathbf{T},.\}$
$E \rightarrow .T, T \rightarrow .T * F, T \rightarrow .F, F \rightarrow .id$
It is possible to have transitions from s_0 on **E, T, F, id** and because, these are the grammar symbols in the items in s_0, where the dot is just before it.

Consider:
$s_1 = \text{goto}(s_0, \mathbf{E}).$
This is simple. Simply let the dot cross **E** wherever the dot is in front of **E.**
This gives:
$s_1 = \{\mathbf{E}' \rightarrow \mathbf{E}., \mathbf{E} \rightarrow \mathbf{E}.+\mathbf{T}\}.$
Note that in both the items, the dot is in front of terminals. So, the "closure" operation will not add further items.
Similarly,
$s_2 = goto(s_0, \mathbf{T}) = \{\mathbf{E} \rightarrow \mathbf{T}., \mathbf{T} \rightarrow \mathbf{T}.*\mathbf{F}\}$
$s_3 = goto(s_0, \mathbf{F}) = \{\mathbf{T} \rightarrow \mathbf{F}.\}$
$s_4 = goto(s_0, \mathbf{id}) = \{\mathbf{E} \rightarrow \mathbf{id}.\}$
We can proceed similarly to get the following:
$s_5 = goto(s_1, +) = \{\mathbf{E} \rightarrow \mathbf{E}+.\mathbf{T}, \mathbf{T} \rightarrow .\mathbf{T}*\mathbf{F},\}$
$T \rightarrow .F, F \rightarrow .id.$
$s_6 = goto(s_2, *) = \{\mathbf{T} \rightarrow \mathbf{T}*.\mathbf{F}, \mathbf{F} \rightarrow .\mathbf{id}\}.$
$s_7 = goto(s_5, \mathbf{T}) = \{\mathbf{E} \rightarrow \mathbf{E}+\mathbf{T}., \mathbf{T} \rightarrow \mathbf{T}.*\mathbf{F}\}.$
$s_8 = goto(s_6, \mathbf{F}) = \{\mathbf{T} \rightarrow \mathbf{T}*\mathbf{F}.\}.$

Additionally:
$goto(s_5, \mathbf{F}) = s_3.$
$goto(s_5, \mathbf{id}) = s_4.$
$goto(s_6, \mathbf{id}) = s_4.$
$goto(s_7, *) = s_6.$
You can clearly visualize the DFA and are encouraged to draw a diagram for the same (see Fig).

We have completed **Step-1** of parser table construction process.



The action/goto tables are extracted directly from the above information in the following manner. First, each state in the above table will be a row in the action/goto tables.
Filling in the entries for row I is done as follows:

- Action[I,c] = shift to state goto (items in state I, c) for terminal c
- Goto[I,c] = goto(items in state I, c) for non-terminal c
- For production number n: A -> α in state I where '.' occurs at the end, Action[I,a] = reduce production n, for all elements a in Follow(A)
- For the added production (augmented production) with the . at the end in state I, Action[I,$] = accept.

The Parsing Table is as below:

State ← Action → GOTO

	id	+	*	$	E	T	F
0	s_4				1	2	3
1		S_5		acc			
2		R_2	S_6	R_2			
3		R_4	R_4	R_4			
4		R_5	R_5	R_5			
5	S_4					7	3
6	S_4						8
7		R_1	S_6	R_1			
8		R_3	R_3	R_3			

6. Consider the grammar $G=\{V,T,S,P\}$; where $V=\{S,A\}$, $T=\{a,b\}$, S is the start variable and $P=\{S\to AS|b,\ A\to SA|a\}$.

(i) Compute the collection of sets of LR(0) item sets for the grammar.
(ii) Construct the SLR parsing table using the SLR algorithm.
(iii) Show all moves allowed by the table from (ii) on the input $abab$.

[WBUT 2015]

Answer:

i) & ii) ***Refer to Question No. 3 of Long Answer Type Questions.***

iii) Augmented Grammar is:

0. $S'\to S$
1. $S\to AS$
2. $S\to b$
3. $A\to SA$
4. $A\to a$

$I_0=\text{CLOSURE}\ (S'\to .S)=\{S'\to .S,\ S'\to .AS,\ S\to .b,\ A\to .SA,\ A\to .a\}$

$I_1=\text{goto}(I_0,S)=\{S'\to S.,\ A\to S.A,\ A\to .SA,\ A\to .a,\ S\to .AS,\ S\to .b\}$

$I_2=\text{go to}\ (I_0,a)=\{A\to a.\}$

$I_3=\text{go to}\ (I_0,b)=\{S\to b.\}$

$I_4=\text{go to}\ (I_0,A)=\{S\to A.S,\ S\to .AS,\ S\to .b,\ A\to .SA,\ A\to .a\}$

$\text{go to}\ (I_1,a)=I_2$

$\text{go to}\ (I_1,b)=I_3$

$I_5=\text{go to}\ (I_1,S)=\{A\to S.A,\ A\to .SA,\ A\to .a,\ S\to .AS,\ S\to .b\}$

$I_6=\text{go to}$

$(I_1,A)=\{A\to SA.,\ S\to A.S,\ S\to .AS,\ S\to .b,\ A\to .SA,\ A\to .a\}$

$I_7=\text{go to}\ (I_4,S)=\{S\to AS.,\ A\to S.A,\ A\to .AS,\ A\to .a\}$

	a	b	\$	S	A
0	$S2$	$S3$		1	4
1	$S2$	$S3$	acc	5	6
2	$r4$	$r4$			
3	$r2$	$r2$	$r2$		
4	$S2$	$S3$		7	6
5	$S2$	$S3$		5	6
6	$S2$ $r3$	$S3$ $r3$		7	4
7	$S2$ $r1$	 $r1$	 $r1$		6

go to $(I_4, a) = I_2$
go to $(I_4, b) = I_3$
I_5 = go to (I_5, S)
I_6 = go to (I_5, A)
go to $(I_5, a) = I_2$
go to $(I_5, b) = I_3$
go to $(I_6, S) = I_7$
go to $(I_6, A) = I_4$
go to $(I_6, a) = I_2$
go to $(I_6, b) = I_3$
go to $(I_7, A) = I_6$
go to $(I_7, a) = I_2$
FIRST $(S) = \{a, b\}$
FIRST $(A) = \{a, b\}$
FOLLOW $(S) = \{a, b, \$\}$
FOLLOW $(A) = \{a, b\}$
The grammar is not SLR.

Configurations

$(0,\ abab\$)$	Initial configuration
$(0a2,\ bab\$)$	Shift
$(0A4,\ bab\$)$	Reduce $A \to a$
$(0A4b3,\ ab\$)$	Shift
$(0A4S7,\ ab\$)$	Reduce $S \to b$
$(0A4S7a2,\ b\$)$	We choose shift
$(0A4S7A6,\ b\$)$	Reduce $A \to a$
$(0A4S7A6b3,\ \$)$	We choose shift
$(0A4S7A6S7,\ \$)$	Reduce $S \to b$
$(0A4S7S?,\ \$)$	Reduce $S \to AS$ but leads to **error**

7. Construct the LR(0) set of items and the SLR parsing table for the following grammar:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

[WBUT 2017]

Answer:

$LR(0)$ set of items and SLR Parsing Table

$$G: \begin{matrix} E \to E+T \mid T \\ T \to T*F \mid F \\ F \to (E) \mid id \end{matrix} \Rightarrow$$

$$G': \begin{matrix} E' \to E \\ E \to E+T \mid T \\ T \to T*F \mid F \\ F \to (E) \mid id \end{matrix}$$ Augmented grammar

$I_0 : E' \to \cdot E$
$E \to \cdot E+T \mid \cdot T$
$T \to \cdot T*F \mid \cdot F$
$F \to \cdot(E) \mid \cdot id$

Goto $(I_0, E) = I_1$
$I_1 : E' \to E \cdot$
$E \to E \cdot + T$

Goto $(I_0, T) = I_2$
$I_2 : E \to T \cdot$
$T \to T \cdot * F$

Goto $(I_0, F) = I_3$
$I_3 : T \to F \cdot$

Goto $(I_0, () = I_4$
$I_4 : F \to (\cdot E)$
$E \to \cdot E+T \mid \cdot T$
$T \to \cdot T*F \mid \cdot F$
$F \to \cdot(E) \mid \cdot id$

Goto $(I_0, id) = I_5$
$I_5 : F \to id \cdot$

Goto $(I_4, E) = I_8$
$I_8 : F \to (E \cdot)$
$E \to E \cdot + T$

Goto $(I_4, T) = I_2$

Goto $(I_4, F) = I_3$

Goto $(I_4, () = I_4$

Goto $(I_4, id) = I_5$

Goto $(I_6, T) = I_9$
$I_9 : E \to E+T$
$T \to T \cdot * F$

Goto $(I_6, F) = I_3$

Goto $(I_6, () = I_4$

Goto $(I_6, id) = I_5$

Goto $(I_7, F) = I_{10}$
$I_{10} : T \to T*F \cdot$

Goto $(I_7, () = I_4$

Goto $(I_7, id) = I_5$

Goto $(I_8, +) = I_6$

Production Rules for G

1) $E \to E+T$
2) $E \to T$
3) $T \to T*F$
4) $T \to F$
5) $F \to (E)$
6) $F \to id$

First and Follow

First $(E') = \{(, id\}$

First $(E) = \{(, id\}$

First $(T) = \{(, id\}$

First $(F) = \{(, id\}$

Follow $(E') = \{\$\}$

Follow $(T) = \{\$, +, *,)\}$

Follow $(F) = \{\$, +, *,)\}$

Follow $(E) = \{\$, +,)\}$

Goto $(I_1, +) = I_6$

$I_6 : E \to E + \cdot T$

$T \to T * F \mid \cdot F$

$F \to \cdot(E) \mid \cdot id$

Goto $(I_2, *) = I_7$

$I_7 : T \to T * \cdot F$

$F \to \cdot(E) \mid \cdot id$

Goto $(I_8,)) = I_{11}$

$I_{11} : F \to (E)\cdot$

Goto $(I_9, *) = I_7$

SLR Parsing Table:

State	Action						Goto		
	id	+	*	(	)	$	*E*	*T*	*F*
0	S_5			S_4			1	2	3
1		S_6				acc			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

8. Consider the following grammar: **[WBUT 2017]**

$S \to CC$

$C \to cC \mid d$

Find the LR(1) set of items.

Answer:

$$G: \begin{matrix} S \to CC \\ C \to cC \mid d \end{matrix} \Big\} \text{ Find } LR(1) \text{ sets of items.}$$

Augmented grammar with an extra starting production is considered first —

$$G': \begin{matrix} S' \to S \\ S \to CC \\ C \to cC \mid d \end{matrix} \Big\} \text{ Augmented grammar}$$

$$\left.\begin{aligned}&\text{First}(S)=\{c,d\}\\&\text{First}(C)=\{c,d\}\\&\text{Follow}(S)=\{\$\}\\&\text{Follow}(C)=\{\$\}\end{aligned}\right\}\text{ First and follow set of the augmented grammar}$$

$LR(1)$ sets of item

$I_0 : S' \to \cdot S, \$$

$S \to CC, \$$

$C \to \cdot cC, c \mid d$

$C \to \cdot d, c \mid d$

Goto $(I_0, S) = I_1$

$S' \to S \cdot, \$$

Goto $(I_0, c) = I_2$

$S \to C \cdot C, \$$

$C \to \cdot cC, \$$

$C \to \cdot d, \$$

Goto $(I_0, c) = I_3$

$C \to c \cdot C, c \mid d$

$C \to \cdot cC, c \mid d$

$C \to \cdot d, c \mid d$

Goto $(I_0, d) = I_4$

$C \to d \cdot, c \mid d$

Goto $(I_2, c) = I_5$

$S \to CC \cdot, \$$

Goto $(I_2, c) = I_6$

$C \to c \cdot C, \$$

$C \to \cdot cC, \$$

$C \to \cdot d, \$$

Goto $(I_2, d) = I_7$

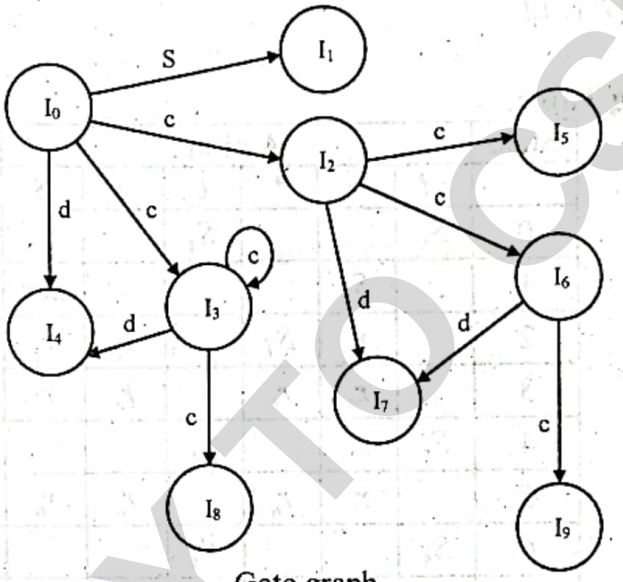
$C \to d \cdot, \$$

Goto $(I_3, c) = I_8$

$C \to cC \cdot, c \mid d$

Goto $(I_3, c) = I_3$

Goto $(I_3, d) = I_4$



Goto graph

LR(1) Set of items

Goto $(I_6, c) = I_9$

$C \rightarrow cC\cdot, \$$

Goto $(I_6, c) = I_6$

Goto $(I_6, d) = I_7$

9. Write short note on Handle pruning. **[WBUT 2012]**

Answer:

A handle of a string is a substring that matches the RHS of a production, and whose reduction to the lhs is one step along the reversal of a rightmost derivation. The process we went through can be viewed as "handle-pruning", where we're pruning the parse tree.

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

$$S=\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \omega$$

input string

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.

SYNTAX DIRECTED TRANSLATION

Multiple Choice Type Questions

1. An annotated parse tree is [WBUT 2006, 2007, 2009]
   a) a parse tree with attribute values shown at the parse tree nodes
   b) a parse tree with values of only some attributes shown at parse tree nodes.
   c) a parse tree without attribute values shown at parse tree nodes
   d) a parse tree with grammar symbols shown at parse tree nodes

Answer: (a)

2. The edges in a flow graph whose heads dominate their tails are called: [WBUT 2008, 2016]
   a) Back edges
   b) Front edges
   c) Flow edges
   d) None of these

Answer: (a)

3. A parse tree showing the values of attributes at each node is called in particular [WBUT 2011, 2018]
   a) Syntax tree
   b) Annotated parse tree
   c) Syntax Direct parse tree
   d) Direct Acyclic graph

Answer: (b)

4. Inherited attribute is a natural choice in [WBUT 2013, 2019]
   a) keeping track of variable declaration
   b) checking for the correct use of L values and R values
   c) both (a) and (b)
   d) none of these

Answer: (c)

5. Which of the following uses only synthesized attributes? [WBUT 2014]
   a) s-attributed grammar
   b) l-attributed grammar
   c) inherited attribute
   d) none of these

Answer: (a)

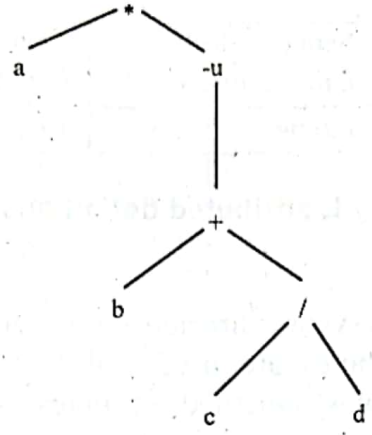
6. Which of the following expression has no l-value? [WBUT 2014]
   a) a(l+1)
   b) a
   c) 7
   d) *a

Answer: (c)

Short Answer Type Questions

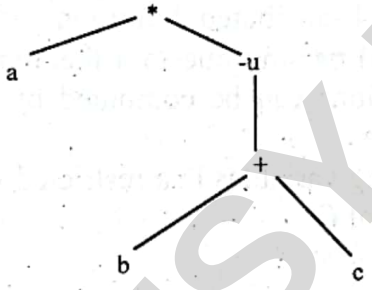
1. Translate the arithmetic expression a * - (b + c / d) into: [WBUT 2007, 2012, 2015]
   Syntax tree

Answer:
The Syntax Tree for the expression a * – (b + c / d) is as shown below:



2. Translate the arithmetic expression $a*-(b+c)$ into: Syntax tree
[WBUT 2008, 2016]

Answer:
The syntax tree for the arithmetic expression a * – (b + c) is:



3. Explain inherited attribute and synthesized attribute for Syntax directed translation with suitable example. **[WBUT 2010, 2015]**

Answer:
For Syntax Directed Translation, compilers extend the normal context-free grammar parse tree to an Annotated Parse Tree where *each* node of the tree is a record with a field for each attribute. The value of an attribute of a grammar symbol at a given parse tree node is defined by a *semantic rule* associated with the production used at that node.
There can be two kinds of attributes:
These attributes are computed from the values of the *attributes of the children nodes.*
These attributes are computed from the values of the *attributes of both the siblings and the children nodes.*
In a syntax directed definition, each production $\mathbf{A} \rightarrow \alpha$ is associated with a set of semantic rules of the form:

$$\mathbf{b = f(c_1, c_2, \ldots, c_n)}$$

where **f** is a function and **b** can be either a synthesized attribute of A and $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the right side of the production, OR an inherited

attribute one of the grammar symbols in the right side of the production), and $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the production.
Example:

Syntax Rule	Semantic Rule	Comment
$D \rightarrow T\ L$	*L.in = T.type*	*L in* is inherited
$T \rightarrow$ **int**	*T.type* ='*integer*'	*T type* is synthesized

4. What do you understand by L-attributed definitions? Illustrate with an example. **[WBUT 2011, 2014]**

Answer:
There are two classes of the syntax directed definitions — S-Attributed **Definitions:** where only synthesized attributes are used and L-Attributed Definitions where, in addition to synthesized attributes, inherited attributes may also be used in a restricted fashion.

Definition: Syntax directed definition is L-Attributed if each inherited attribute of X_j in a production $A \rightarrow X_1 X_2 \ldots . X_j \ldots . X_n$, depends only on: i. The attributes of the symbols to the left of X_j, i.e., $X_1, X_2, \ldots, X_{j-1}$, and ii. The inherited attributes of A .

Every S-attributed definition is an L-attributed definition and an L-attributed definition is more conducive to LL (top-down) parsing due to a theorem which says that "Inherited attributes in L-Attributed Definitions can be computed by a Pre-order traversal of the parse-tree".

Consider the grammar for declaring variables in a restricted version (only 'char' and 'int' as allowed types and no pointers) of C:

$S \rightarrow T\ L$

$L \rightarrow L$ id | id

$T \rightarrow$ char | int

If each 'id' has to have the attribute type, then the second rule must be associated with the definition:

id.type = L.type

which is an example of an L-attributed definition.

5. Define grammar. What do you mean by ambiguity in grammar? Illustrate with suitable example. What is the necessity to generate parse tree? **[WBUT 2012]**

Answer:

1st Part:

Grammar is a specification for the syntax of a programming language.

2nd Part:

A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.

Example:

$G = (\{S\}, \{a + b, +, *\}, P, S)$ where P consists of $S \rightarrow S + S \mid S * S \mid a \mid b$

The String a + a * b can be generated as:

$S \rightarrow \underline{S} + S$
$\rightarrow a + \underline{S}$
$\rightarrow a + \underline{S} * S$
$\rightarrow a + a * \underline{S}$
$\rightarrow a + a * b$

$S \rightarrow \underline{S} * S$
$\rightarrow \underline{S} + S * S$
$\rightarrow a + \underline{S} * S$
$\rightarrow a + a * \underline{S}$
$\rightarrow a + a * b$

Thus, this Grammar G is Ambiguous.

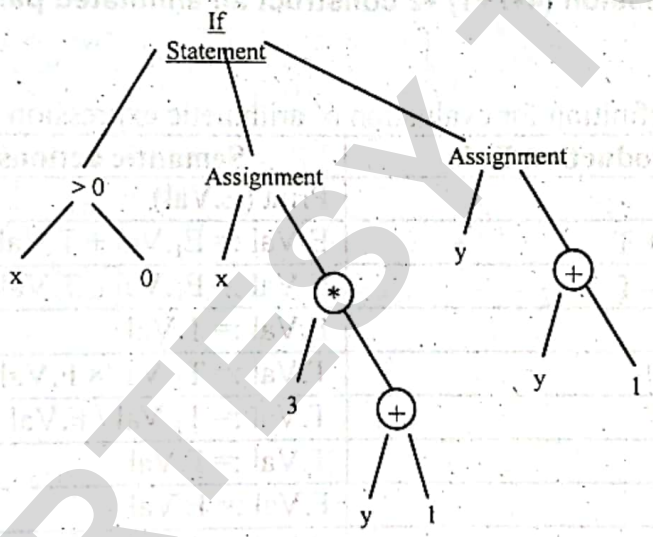
3rd Part:

A parse tree is a data-structure generated by the parse which captures the 'meaning' of the input source code as per the language that the compiler caters to. Subsequent stage of the compiler like syntax-generated translation can convert the parse tree into the target code which can then be optimized further.

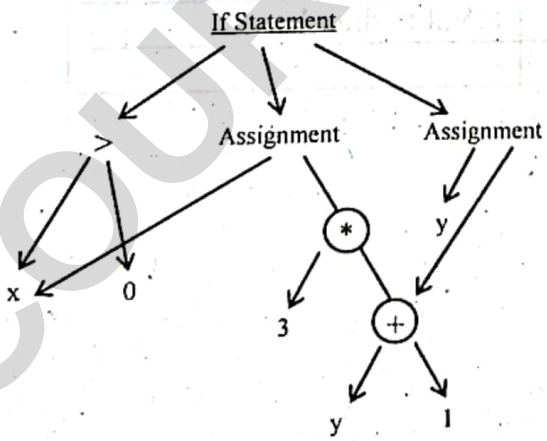
6. Draw a Syntax tree and DAG from the following expression. If $x > 0$ then $x = 3 * (y + 1)$ else $y = y + 1$ **[WBUT 2013]**

Answer:

Syntax Tree



DAG



7. When a grammar is called ambiguous? Is there any technique to remove ambiguity? **[WBUT 2013]**

OR,

What is Ambiguous Grammar? **[WBUT 2016]**

Answer:

A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.

There are several ways to remove ambiguity:

(A) Fix grammar so it is not ambiguous. (Not always possible or reasonable or possible.)

(B) Add grouping/precedence rules.

(C) Use semantics: choose parse that makes the most sense.

Grouping/precedence rules are the most common approach in programming language applications.

Invoking semantics is more common in natural language applications.

Fixing the grammar is less often useful in practice, but neat when you can do it.

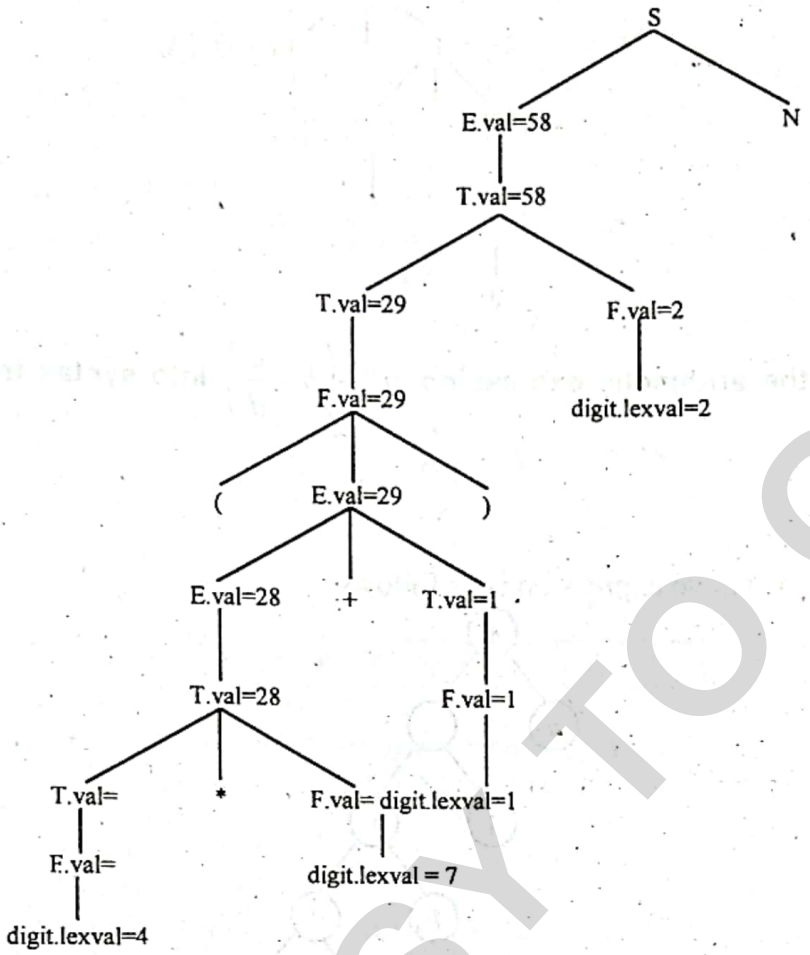
8. For the input expression (4*7+1) *2 construct an annotated parse tree. **[WBUT 2014]**

Answer:

The syntax directed definition for evaluation of arithmetic expression is given as:

Production Rule	Semantic actions
$S \rightarrow EN$	Print (E.Val)
$E \rightarrow E_1 + T$	E.Val := E_1.Val + T.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
$T \rightarrow T_1 / F$	T.Val := T_1.Val / F.Val
$T \rightarrow F$	T.Val := F.Val
$F \rightarrow (E)$	F.Val := E.Val
$F \rightarrow$ digit	F.Val := digit.lexval
$N \rightarrow :$	-

The annoted parse tree can be drawn as follows:



9. Generate an annotated parse tree for the string "3+2−4" using the grammar

$E \rightarrow E+T \mid E-T \mid T \quad T \rightarrow 0|1|2|.....|9$ **[WBUT 2015]**

Answer:

Grammar

$E_1 \rightarrow E_2 + T$

$E_1 \rightarrow E_2 - T$

$E \rightarrow T$

$T \rightarrow 0|1|2|.....|9$

Semantic Rule:

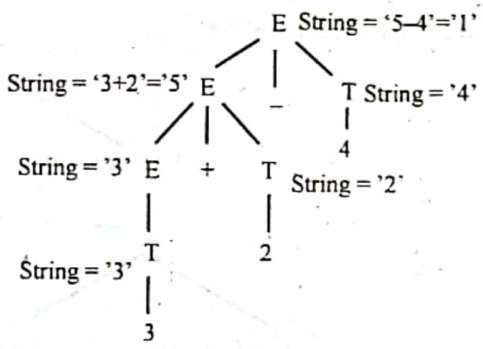
E_1.String = E_1.String ||T.String || '+'

E_1.String = E_1.String ||T.String || '–'

E_1.String = T.String

T.String = 0.String | 1.String|....|9.String

For the input String '3+2–4' we have the denoted parse tree below with the corresponding values for the string attribute.

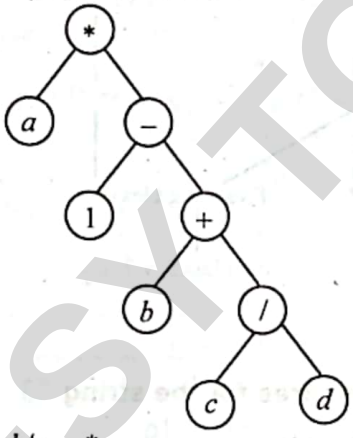


10. Translate the arithmetic expression $a*-\left(b+\frac{c}{d}\right)$ into syntax tree and postfix notation. **[WBUT 2015]**

Answer:

$a*-(b+c/d)$

Syntax tree for the above expression is as follows:



and the postfix notation in $a1bcd/+-*$

11. a) Design a direct acyclic graph for the string: **[WBUT 2017]**

$a+a*(b-c)+(b-c)*d$.

Answer:

Refer to Question No. 2(2nd Part) of Long Answer Type Questions.

b) What are the two types of attributes that are associated with a grammar symbol? **[WBUT 2017]**

Answer:

Two types of attributes are synthesized attribute and inherited attribute.

The **attribute** which takes data values from its child nodes, is called **synthesized attribute**. These are also called s-**attributed** production. The **attribute** which takes values form parents or sibling nodes are called inherited **attributes**. The production rule having inherited **attribute** are called L-**attributed** productions.

12. What is ambiguity in grammar? Justify whether the grammar is ambiguous or not. A→AA|(A)|a **[WBUT 2018]**

Answer:

1st Part: ***Refer to Question No. 5(2nd Part) of Short Answer Type Questions.***

2nd Part:

Given Grammar is: $A \rightarrow A\,A \mid (A) \mid a$

$A \rightarrow \underline{A}\,A$
$\rightarrow A\,A\,\underline{A}$
$\rightarrow \underline{A}\,A\,a$
$\rightarrow a\,\underline{A}\,a$
$\rightarrow a\,\underline{A}\,A\,a$
$\rightarrow a\,(\underline{A})\,A\,a$
$\rightarrow a\,(a)\,\underline{A}\,a$
$\rightarrow a\,(a)\,a\,a$

$A \rightarrow \underline{A}\,A$
$\rightarrow a\,\underline{A}$
$\rightarrow a\,\underline{A}\,A$
$\rightarrow a\,A\,A\,\underline{A}$
$\rightarrow a\,\underline{A}\,A\,a$
$\rightarrow a\,(\underline{A})\,A\,a$
$\rightarrow a\,(a)\,\underline{A}\,a$
$\rightarrow a\,(a)\,a\,a$

Thus, this Grammar A is ambiguous.

Long Answer Type Questions

1. Suppose a robot can be instructed to move one step east, north, west or south from its current position. A sequence of such instruction is generated by the following grammar:

$Seq \rightarrow Seq_1$ instr | begin

Instr → east | north | west | south

i) Construct a syntax directed definition to translate an instruction sequence into a robot position.

ii) Draw a parse tree for: begin west south. **[WBUT 2010]**

Answer:

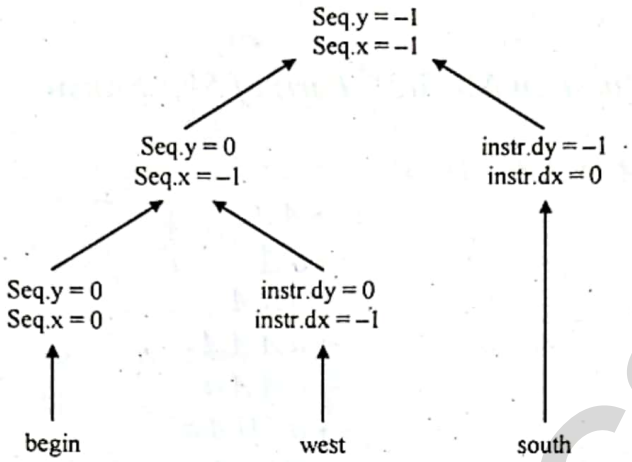
i) We shall use two attributes, $Seq.x$ and $Seq.y$, to keep track of the position resulting from an instruction sequence generated by the nonterminal Seq. initially, Seq generates **begin** and $Seq.x$ and $Seq.y$ are both initialized to 0.

The change in position due to an individual instruction derived from non-terminal $instr$ is given by attributes $instr.dx$ and $inxtr.dy$.

A syntax-directed definition for translating an instruction sequence into a robot position is shown below:

PRODUCTION	SEMANTIC RULES
$Seq \rightarrow$ **begin**	$Seq.x = 0;\ Seq.y = 0;$
$Seq \rightarrow Seq_1\ instr$	$Seq.x = Seq_1.x + instr.dx;\ Seq.y = Seq_1.y + instr.dy;$
$instr \rightarrow$ **east**	$instr.dx = 1;\ instr.dy = 0;$
$instr \rightarrow$ **north**	$instr.dx = 0;\ instr.dy = 1;$
$instr \rightarrow$ **west**	$instr.dx = -1;\ instr.dy = 0;$
$instr \rightarrow$ **south**	$instr.dx = 0;\ instr.dy = -1;$

ii) The annotated parse tree for the expression is given below



2. What are the main contributions of Syntax Directed Translation in Compiler? Design a Dependency Graph and Direct acyclic graph for the string

a+ a * (b – c) + (b – c) * d **[WBUT 2010, 2011]**

Answer:

1st Part:

For programming language grammars, the semantics of the input sentence (i.e. the source ode of the program being compiled) should get captured as a body of executable code (in compilers) or as direct execution, possibly in stages, as in an interpreter. Syntax-directed translation refers to a method of compiler implementation where the source language translation is completely driven by the parser. In other words, the parsing process and parse trees are used to direct semantic analysis and the translation of the source program. The principle of Syntax Directed Translation states that *the meaning of an input sentence is related to its syntactic structure (i.e. to its Parse-Tree)*.Syntax Directed Translation deals with formalisms for specifying translations for the source (programming) language constructs guided by the context-free grammar that describes the language. The complete sematic information (i.e., the ``object code'') for the given ``sentence'' (i.e., a program written in the language) *is incrementally computed from syntax analysis*.

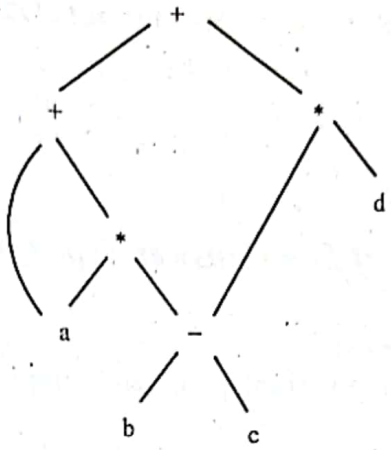
For programming language compilers, the Syntax Directed Translation phase additionally checks and validates context-sensitive conditions like:

- Variables are declared before used.
- Type compatibility.
- Formal/actual parameter agreement in procedures.
- Resolve overloaded symbols.
- Others, depending upon that particular language

2nd Part:

For the given expression, the Dependency Graph is simply the parse tree with every edge directed upwards in terms of dependency. This is because, because only one attribute, namely *E.val* is required in every (non-leaf) node.

The DAG for the expression is as given below:



3. a) What do you mean by input buffering? **[WBUT 2014]**
b) How is input buffering implanted?
c) What problem can arise implementing input buffering? Give a suitable example.
d) What is sentinel?
e) What is its use?

Answer:

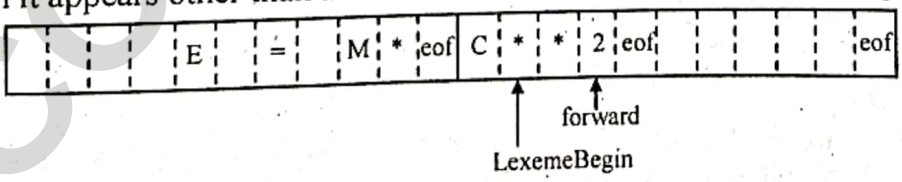
a) & b) ***Refer to Question No. 5(c) of Long Answer Type Questions.***

c) In single buffering, a lexeme may cross the buffer boundary which requires the buffer to be refilled, thus overwriting the part of the lexeme that was in the buffer before reloading.

Example: Suppose the last 4 characters of the buffer are 'm', 'y', 'v' and 'a'; the new lexeme starts at 'm' and "myva" is possibly the start of a new identifier "myvar" (say). But the entire lexeme was not found in the buffer, causing it to be reloaded. The freash buffer has 'r' and '=' as the first two characters, clearly meaning that "myvar" was indeed the last lexeme. But while loading th buffer, the "myva" part of the identifier is overwritten.
The above problem is solved by a two buffer scheme. In that case, the problem can arise only if a lexeme is bigger than the size of the buffer which is extremely unlikely.

d) A Sentinel is a special character that cannot be part of the source program. It is added at each buffer end to test normally we use 'eof' as the sentinel. Actually sentinel saves time checking for the ends of buffers.

e) This is used for speeding-up the lexical analyzer. It retains the role of entire input end. When it appears other than at the end of a buffer it means that the input is at an end



4. a) What are the main contributions of Syntax Directed Translation in Compiler?
b) Mention different loop optimization techniques. Optimize the following code:

```
do{
        item = 10;
        x = x + item;
}while (value<50);
```

[WBUT 2018]

Answer:

a) *Refer to Question No. 2 of Long Answer Type Questions.*

b) Since `item=10;` and `x=x+ikn;`
are both independent of the loop variable, we can being them out of the loop.
Optimized Code:

```
item=10;
x=x+ikn;
do {
} while (value<50);
```

5. Write short note on the following:

a) Inherited attributes **[WBUT 2008]**
b) Dependency graph **[WBUT 2009, 2015]**
c) Input buffering **[WBUT 2009, 2011]**
d) L-attributed definitions **[WBUT 2009, 2015]**
e) LALR **[WBUT 2018]**

Answer:

a) Inherited attributes:

Inherited attributes are attributes that are passed to a rule, as opposed to synthesized attributes, which are returned from a rule. These attributes are computed from the values of the attributes of both the siblings and the children nodes of Annotated Parse Trees. Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which it appears. For example, we can use an inherited attribute to keep track of whether an identifier appears on the left or the right side of an assignment in order to decide whether the address or the value of the identifier is needed. Although it is always possible to rewrite a syntax directed definition to use only synthesized attributes, it is often more natural to use definitions with inherited attributes. For example, if in a grammar, the production rule D→TL is associated with the semantic rule L:in = T:type (this is a case of variable declaration in C/C++ where L lists a list of identifiers), the types of the identifiers are inherited from the type declaration encoded in the non-terminal T. The declared type is percolated down the syntax tree headed by L down to each identifier.

b) Dependency Graph:

The next logical stage in the translation process is to create a data structure called the Dependency Graph. The Dependency Graph is a superimposition on the annotated parse tree and sets up the dependencies between the attributes of the nodes.
A Dependency Graph is:

- Directed Graph.
- Shows interdependencies between attributes of the nodes of the annotated parse tree.

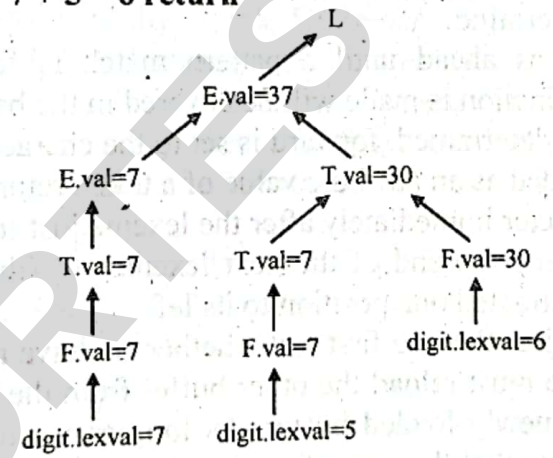
The following steps are carried out to construct a Dependency Graph:

```
for each node n in the parse tree {
for each attribute a of the grammar symbol at n {
construct a node in the dependency graph for a
}
}
for each node n in the parse tree {
for each semantic rule b = f(c1, c2, ... ,cn)
for rule used at n {
for i= 1 to n {
construct an edge from the node for ci to the
node for b
}
}
}
```

The Dependency graph for a valid and translatable expression should be a Directed Acyclic Graph (or DAG). Otherwise, some attribute of a node would have circular dependence on itself which is ridiculous. To evaluate the attributes, first a topological sort is carried out on the DAG and that can now be used as a guide for the order of attribute evaluation. The examples given below should make the ideas clear.

Example:

Dependency Graph for 7 + 5 * 6 return



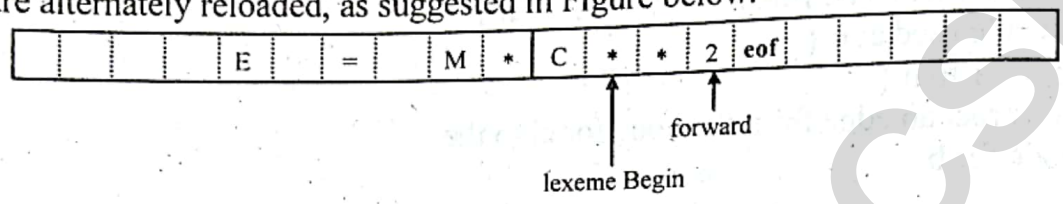
c) Input Buffering:

We should examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. There are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for **id**. In C, single-character operators like -, =, or < could also be the beginning of a two-character

operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large lookaheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

Buffer Paris

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded, as suggested in Figure below.



Using a pair of input buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained:

1. Pointer lexeme Begin marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer forward scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, forward is set to the character at its right end. Them, after the lexeme is recorded as an attribute value of a token returned to the parser, lexeme Beg in is set to the character immediately after the lexeme just found. In the above figure, we see forward has passed the end of the next lexeme, ** (the Fortran exponentiation operator), and must be retracted one position to its left.

Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

d) L-attributed definitions:

L-Attributed Definitions contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them.

Definition: A syntax directed definition is L-Attributed if each inherited attribute of Xj in a production $A \rightarrow X_1X_2 \ldots\ldots X_j \ldots\ldots X_n$, depends only on:

1. The attributes of the symbols to the left of X_j, i.e., $X_1;X_2;$: : : $;X_j$-1, and
2. The inherited attributes of A.

Clearly, every S-attributed definition is an L-attributed definition. Also, an Lattributed definition is more conducive to LL (top-down) parsing.

Theorem: Inherited attributes in L-Attributed Definitions can be computed by a Pre-order traversal of the parse-tree.

Bottom-up Evaluation of L-attributed Grammars

For grammars with L-attributed definitions, special evaluation algorithms must be designed in case of bottom-up parsers. Such algorithms can handle most LR(1) grammars. The basic idea is to are take all translation actions to the right end of the production. The key observation here is that when a bottom-up parser reduces by the rule A!XY, it removes X and Y from the top of the stack and replaces them by A. Since the synthesized attributes of X is on the top of the stack, they thus can be used to compute attributes of Y. In real implementations, information contained deeper in the stack can be floated up by using special markers to mark the production we are currently in. Markers can also be used to perform error checking and other intermediate semantic actions.

e) LALR:

In computer science, an **LALR parser** or **Look-Ahead LR parser** is a simplified version of a canonical LR parser, to parse (separate and analyze) a text according to a set of production rules specified by a formal grammar for a computer language. ("LR" means left-to-right, rightmost derivation)

the LALR parser was invented by Frank DeRemer in his 1969 PhD dissertation. **Practical Translators for LR(k) languages**, in his treatment of the practical difficulties at that time of implementing LR(1) parsers. He showed that the LALR. Despite this weakness, the power of the LALR parser is sufficient for many mainstream computer languages, including Java, though the reference grammars for many languages fail to be LALR due to being ambiguous.

The original dissertation gave no algorithm for constructing such a parser given a formal grammar. The first algorithms for LALR parser generation were published in 1973. In 1982, DeRemer and Tom Pennello published an algorithm that generated highly memory-efficient LALR parsers. LALR parsers can be automatically generated from a grammar by an LALR parser generator such as Yacc or GNU Bison. The automatically generated code may be augmented by hand-written code to augment the power of the resulting parser.

The LALR(1) parser is less powerful than the LR(1) parser, and more powerful than the SLR(1) parser, though they all use the same production rules. The simplification that the LALR parser introduces consists in merging rules that have identical **kernel item sets**, because during the LR(0) state-construction process the lookaheads are not known. This reduces the power of the parser because not knowing the lookahead symbols can confuse the parser as to which grammar rule to pick next, resulting in **reduce/reduce conflicts**. All conflicts that arise in applying a LALR(1) parser to an unambiguous LR(1) grammar

are reduce/reduce conflicts. The SLR(1) parser performs further merging, which introduces additional conflicts.

The standard example of an LR(1) grammar that cannot be parsed with the LALR(1) parser, exhibiting such a reduce/reduce conflict, is:

```
S → a E c
  → a F d
  → b F c
  → b E d
E → e
F → e
```

In the LALR table construction, two states will be merged into one state and later the lookaheads will be found to be ambiguous. The one state with lookaheads is:

```
E → e. (c,d)
F → e. (c,d)
```

An LR(1) parser wil create two different states (with non-conflicting lookaheads), neither of which is ambiguous. In an LALR parser this one state has conflicting actions (given lookahead c or d, reduce to E or F), a "reduce/reduce conflict"; the above grammar will be declared ambiguous by a LALR parser generator and conflicts will be reported.

To recover, this ambiguity is resolved by choosing E, because it occurs before F in the grammar. However, the resultant parser will not be able to recognize the valid input sequence `b e d`, since the ambiguous sequence `e c` is reduced to `(E → e) c`, rather than the correct `(F → e) c`, but `b E c` is not in the grammar.

TYPE CHECKING

Multiple Choice Type Questions

1. Which of the following is not true about dynamic type checking? **[WBUT 2006, 2007, 2008, 2009, 2011, 2018]**

a) Type checking is done during the execution
b) It increases the cost of execution
c) All the type errors are detected
d) None of these

Answer: (d)

2. Which one of the following is not true about dynamic checking? **[WBUT 2010]**

a) It increases the cost of execution
b) Type checking is done during execution
c) All the type error are detected
d) None of these

Answer: (d)

3. Type checking is done normally during **[WBUT 2015, 2017]**

a) lexical analysis
b) syntax analysis
c) syntax directed translation
d) code generation

Answer: (c)

Short Answer Type Questions

1. What is type checking? Differentiate between Dynamic and Static Type checking. **[WBUT 2010, 2016]**

Answer:

For many programming languages, the compiler must check that the source program follows *Type Rules* of the language, i.e., both the syntactic and semantic conventions of the source language. This is called **static checking**. Type checking is targeted towards:

Compiler can detect meaningless or invalid code.

Provide useful compile-time information like memory alignment that can result in more efficient machine instructions.

Types can serve as a form of documentation.

Types allow programmers to think about programs at a higher level than the bit or byte.

A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time. Examples of languages that use static typing include C, C++, C#, Java, Fortran, Pascal, and Haskell.

A programming language is said to use dynamic typing when type checking is performed at run-time (also known as "late-binding") as opposed to at compile-time. Examples of languages that use dynamic typing include Javascript, Lisp, Perl, PHP, Python, Ruby, and Smalltalk.

RUNTIME ENVIRONMENT

Multiple Choice Type Questions

1. Symbol table can be used for [WBUT 2010]
   a) checking type compatibility
   b) suppressing duplicate error messages
   c) storage allocation
   d) all of these

Answer: (d)

2. Number of states of FSM required to simulate behaviour of a computer with a memory capable of storing m words, each of length n [WBUT 2019]
   a) $m \times 2^n$
   b) 2^{mn}
   c) $2^{(m+n)}$
   d) all of these

Answer: (b)

Short Answer Type Questions

1. What is activation record? Explain clearly the components of an activation record. [WBUT 2012, 2015]

Answer:

Refer to Question of No. 1 of Long Answer Type Questions.

2. What are the various data structures used for symbol table construction? [WBUT 2013]

Answer:

Linked List, Sorted and unsorted list, search tree, binary tree, hash table are the data structures used for symbol table.

Long Answer Type Questions

1. What is an activation record? [WBUT 2006, 2007, 2008, 2016]
When and why are those records used? [WBUT 2006, 2008, 2016]
List different fields of an activation record and state the purpose of those fields. [WBUT 2006, 2007, 2008]

OR,

What is activation record? Explain clearly the components of an activation record. [WBUT 2014]

Answer:

1st Part:

An Activation Record (also known as a "stack frame" of a "call stack") is a data structure containing the variables belonging to one particular scope (e.g. a procedure body), as well as links to other activation records.

2nd Part:

These variables may be the actual parameters passed on to the procedure or variables declared in the scope of the procedure. Activation records are usually created on the call stack on entry to a block and destroyed on exit.

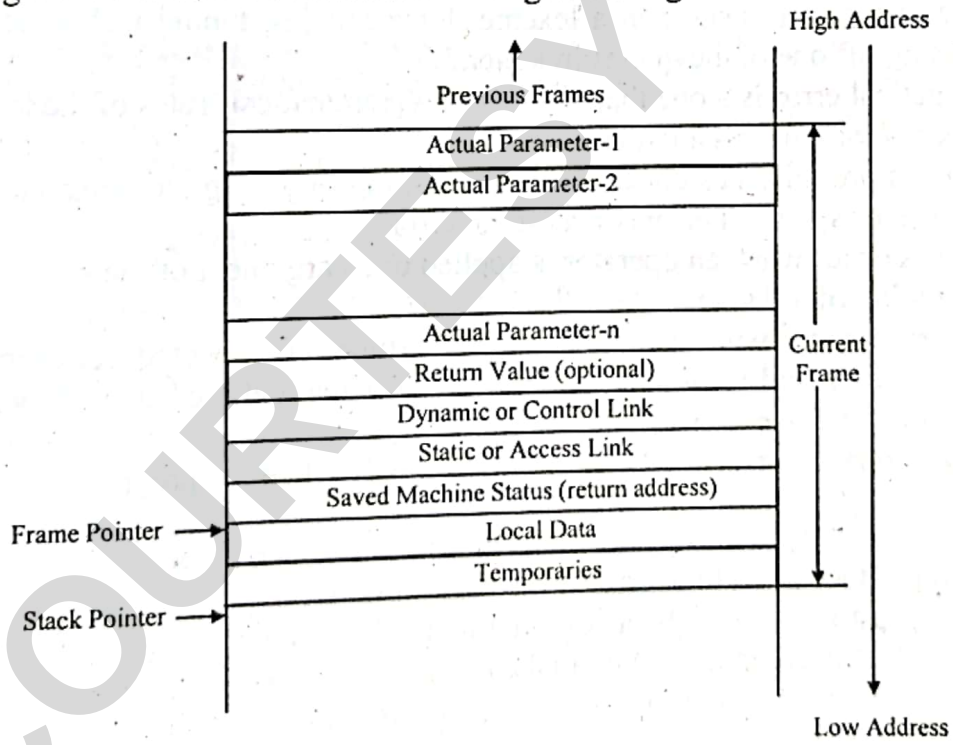
Variables in the current scope are accessed via the frame pointer which points to the current activation record. Variables in an outer scope are accessed by following chains of links between activation records. There are two kinds of link— the static link and the dynamic link.

Given below is a summary of functions of the activation record, depending on the language, operating system, and machine environment, etc.:

- Local data storage
- Parameter passing
- Evaluation stack (for example, temporary variables) for arithmetic or logical operations
- Pointer to current instance in object-oriented languages (e.g., this in C++)
- Enclosing subroutine context
- Other return states besides the return address — environment and/or machine specific

3rd Part:

A typical organization of an activation record is given in figure below:



Before the run-time environment creates the frame for the callee, it needs to allocate space for the callee's arguments. These arguments belong to the caller's frame, not to the callee's frame. There is a frame pointer (called FP) that points to the beginning of the

frame. The stack pointer points to the first available byte in the stack immediately after the end of the current frame (the most recent frame).
There are many variations to this theme. Some compilers use displays to store the static chain instead of using static links in the stack. A display is a register block where each pointer points to a consecutive frame in the static chain of the current frame. This allows a very fast access to the variables of deeply nested functions.
Another important variation is to pass arguments to procedures using registers.
When a procedure starts running, the frame pointer and the stack pointer contain the same address. While the procedure is active, the frame pointer, points at the top of the parameters and return address. The stack pointer may be changed its value while a procedure is active but the frame pointer does not change its value while a procedure is active. The variables will always be the same distance from the unchanging frame pointer.

2. a) What is symbol table? How is it implemented? **[WBUT 2016]**
b) Explain Syntax and Semantic error.
Answer:
a) ***Refer to Question No. 4(a) of Long Answer Type Questions.***

b) Errors are either syntactic or semantic:
Syntax errors are in the program text; they may be either lexical or grammatical:
(a) A lexical error is a mistake in a lexeme, for examples, typing technique instead of then, or missing off one of the quotes in a literal.
(b) A grammatical error is a one that violates the (grammatical) rules of the language, for example if x = 7 y : = 4 (missing then).
Semantic errors are mistakes concerning the meaning of a program construct; they may be either type errors, logical errors or run-time errors:
(a) Type errors occur when an operator is applied to an argument of the wrong type, or to the wrong number of arguments.
(b) Logical errors occur when a badly conceived program is executed, for example: while x = y do ... when x and y initially have the same value and the body of loop need not change the value of either x or y.
(c) Run-time errors are errors that can be detected only when the program is executed, for example:

```
var x : real; read(x); writeln(1/x)
```

which would produce a run time error if the user input 0.
Syntax errors must be detected by a compiler and at least reported to the user (in a helpful way). If possible, the compiler should make the appropriate correction(s). Semantic errors are much harder and sometimes impossible for a computer to detect.

3. Discuss the concepts of parameter pass mechanisms. **[WBUT 2019]**

Answer:

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism.

Basic Terminology:

1. R- value: The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if its appear on the right side of the assignment operator. R-value can always be assigned to some other variable.

2. L-value: The location of the memory (address) where the expression is stored is known as the l-value of that expression. It always appears on the left side if the assignment operator.

Formal Parameters:

Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

Actual Parameters:

Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

```
fun_one()
{
  int actual_parameter = 10;
   call fun_two(int actual_parameter);
}
   fun_two(int formal_parameter)
{
   print formal_parameter;
}
```

Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

4. Write Short Note on following:

a) Symbol table organization **[WBUT 2006, 2007, 2016]**

OR,

Symbol Table **[WBUT 2009, 2010, 2011, 2012, 2013, 2015, 2018]**

OR,

Symbol table manager **[WBUT 2019]**

b) Activation record **[WBUT 2010, 2011, 2018]**

Answer:

a) Symbol table organization:

A symbol table is a data structure, where information about program objects is gathered. It is used in both the lexical analysis and code generation phases. The symbol table is built up during the lexical (and partially during) syntactic analysis. It provides help for other phases during compilation like during Semantic analysis in resolving type conflict,

during Code generation in evaluating how much and what type of run-time space is to be allocated, during Error handling to produce the most appropriate error message, etc. Symbol table management refers to the symbol table's storage structure, its construction in the analysis phase and its use during the whole compilation.

Requirements for symbol table management (or symbol table organization) include:

- quick insertion of an identifier
- quick search for an identifier
- efficient insertion of information (attributes) about an identifier
- quick access to information about a certain identifier
- Space-efficiency and time-efficiency

There may be several possible implementations of a Symbol Table:
Unordered list: Appropriate for a very small set of variables, that too for a simple symbol table.
Ordered linear list: Entries are sorted/indexed by symbol names. Insertion is expensive time-wise, but implementation is relatively easy.
Binary search tree: A variation of a sorted list with O(log n) time per operation for n variables.
Hash table: The most commonly used implementation with very efficient insertion/lookup, provided the memory space is adequately larger than the number of variables.

b) Activation Record:
Refer to Question No. 1 of Long Answer Type Questions.

INTERMEDIATE CODE GENERATION

Multiple Choice Type Questions

1. Which of the following is not an intermediate code form? [WBUT 2010, 2012]
   a) Postfix notation
   b) Syntax trees
   c) Three address codes
   d) Quadruples

Answer: (c)

2. Three-address code involves [WBUT 2013]
   a) exactly 3 addresses
   b) at most 3 addresses
   c) no unary operator
   d) none of these

Answer: (b)

3. Which of the following is not an intermediate code form? [WBUT 2015]
   a) Quadruples
   b) Triples
   c) Abstract syntax tree
   d) Indirect triples

Answer: (c)

Short Answer Type Questions

1. Translate the arithmetic expression $a * -(b+c)$ into: [WBUT 2008, 2016]

a) Three-address code
b) Postfix notation

Answer:

a) The three address code for the arithmetic expression a * –(b + c) is:

t1 = b + c
t2 = –t1
t3 = a * t2

b) The postfix expression for the arithmetic expression $a * -(b+c)$ is:

abc+-*

2. Explain the following terms with the example given below: [WBUT 2008, 2016]
   a) Quadruples
   b) Triples
   c) Indirect Triples

Example: a := b * (c + d / b) - (e * f)

Answer:

A three-address code, i.e., a collection of several three-address statements, can be represented in three forms — quadruples, triples and indirect triples.

The set of three-address codes for the example is:

```
t1 := d / b
t2 := c + t1
t3 := b * t2
t4 := - t3
t5 := e * f
t6 := t4 - t5
a := t6
```

A quadruple is a record structure with four fields — 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 arg 1, z in arg 2 and x in result. Statements with unary operators like **x := –y** or **x := y** do not use arg2. Conditional and unconditional jumps put the target label in result. The quadruples for the example code is:

	op	Res	arg1	arg2
(101)	t1	/	d	b
(102)	t2	+	c	t1
(103)	t3	*	b	t2
(104)	t4	uniminus	t3	
(105)	t5	*	e	f
(106)	t6	–	t4	t5
(107)	a	:=	t6	

If three-address statements are represented by records with only three fields – op, arg1 and arg2, we call them triples. The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table (for programmer defined names or constants) or pointers into the triple structure (for temporary values). The triples for the given code is:

	op	arg1	arg2
(101)	/	d	b
(102)	+	c	(101)
(103)	*	b	(102)
(104)	uniminus	(103)	
(105)	*	e	f
(106)	–	(104)	(105)
(107)	:=	a	(106)

Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves.

This implementation is naturally called indirect triples. The indirect triples for the given code is:

(1)	(101)
(2)	(102)
(3)	(103)
(4)	(104)
(5)	(105)
(6)	(106)
(7)	(107)

	op	arg1	arg2
(101)	/	d	b
(102)	+	c	(101)
(103)	*	b	(102)
(104)	uniminus	(103)	
(105)	*	e	f
(106)	–	(104)	(105)
(107)	:=	a	(106)

3. Compare quadruples, triples and indirect triples. **[WBUT 2013]**

$$x = (a+b)*-c/d$$

Represent this expression in quadruples, triples and indirect triples form.

Answer:

1st Part:

Three address code instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are called

Quadruples: A quadruple (or just "quad') has four fields, which we call op, arg,, arg2, and result

Triples: A triple has only three fields, which we call op, arg1, and arg2. the DAG and triple representations of expressions are equivalent

Indirect Triples: consist of a listing of pointers to triples, rather than a listing of triples themselves.

The benefit of Quadruples over Triples can be seen in an optimizing compiler, where instructions are often moved around.

With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change. With triples, the result of an operation is referred to by its position, so moving an instruction may require to change all references to that result. This problem does not occur with indirect triples.

2nd Part:

Quadruples

	op	res	Arg1	Arg2
(101)	T1	+	a	b
(102)	T2	/	c	d
(103)	T3	uniminus	T2	
(104)	T4	*	T1	T3

Triples

	op	Arg1	Arg2
(101)	+	a	b
(102)	/	c	d
(103)	uniminus	(102)	
(104)	*	(101)	(103)

Indirect triples

	op	Arg1	Arg2
(101)	+	a	b
(102)	/	c	d
(103)	uniminus	(102)	
(104)	*	(101)	(103)

	Statement
(1)	(101)
(2)	(102)
(3)	(103)
(4)	(104)

4. Generate three address code for the following program segment: **[WBUT 2016]**

while (a < c and b > d) do
if a = 1 then c = c + 1;
else
while a < = d do
a = a+ 3;

Answer:

100: if (a <c and b<d) goto 102
101: goto 112
102: if a==1 goto 104
103: goto 107
104: t1: = c+1
105: c:= t1
106: goto 100
107: if (a<= d) goto 109
108: goto 100
109: t2: = a+3
110: a:= t2
111: goto 100

Long Answer Type Questions

1. a) Distinguish between quadruples, triples and indirect triples for the expression:
a = b * - c + b * - c
b) Translate the arithmetic expression a * – (b + c / d) into: **[WBUT 2007, 2009]**
i. Postfix notation **[WBUT 2007, 2012]**
ii. 3-address code.
c) Generate machine code for the following instruction:
X = a / – (b * c) - d
Assume 3 registers are available. **[WBUT 2007, 2015]**

Answer:

a) Quadruple:

	res	op	arg1	arg2
(101)	t1	uniminus	c	
(102)	t2	*	b	t1
(103)	t3	+	t2	t2

Triple:

	op	arg1	arg2
(101)	uniminus	c	
(102)	*	b	(101)
(103)	+	(102)	(102)

Indirect Triple:

	Statement
(1)	(101)
(2)	(102)
(3)	(103)

	op	arg1	arg2
(101)	uniminus	c	
(102)	*	b	(101)
(103)	+	(102)	(102)

b) (i) The Postfix notation for the expression a * - (b + c / d) is **abcd/+-*.**

(ii) The 3-Address code for the expression a * - (b + c / d) is:

t1 = c / d
t2 = b + t1
t3 = -t2
t4 = a * t3

c) The machine code for X = a / - (b * c) - d is:

```
MOV R1, a
MOV R2, b
MUL R2, c
NEG R2
DIV R1, R2
MOV R3, d
SUB R1, R3
MOV X, R1
```

2. a) What are the differences among Quadruples, Triples and Indirect Triples? **[WBUT 2009, 2017]**

OR,

Differentiate Quadruple, Triples and Indirect triples with example. **[WBUT 2010]**

b) Generate machine code for the followings instruction: **[WBUT 2009]**

v = a + (b* c) – d.

Answer:

a) The difference between triples and quadruples may be regarded as a matter of how much indirection is present in the representation. When we ultimately produce target code, each name, temporary or programmer-defined, will be assigned some runtime memory location. This location will be placed in the symbol-table entry for the datum. Using the quadruple notation, a three-address statement defining or using a temporary can immediately access the location for that temporary via the symbol table. A more important benefit of quadruples appears in an optimizing compiler, where statements are often moved around. Using the quadruple notation, the symbol table interposes an extra degree of indirection between the computation of a value and its use. If we move a statement computing x, the statements using x require no change. However, in the triples declaration. moving a statement that defines a temporary value requires us to change all references to that statement in the arg1 and arg2 arrays. This problem makes triples difficult to use in an optimizing compiler. Indirect triples present no such problem. A statement can be moved by reordering the statement list. Since pointers to temporary values refer to the right-side arg1 and arg2 arrays, which are not changed, none of those pointers need be changed. Thus, indirect triples look very much like quadruples as far as their utility is concerned. The two notations require about the same amount of space and they are equally efficient for reordering of codes with ordinary triples. Allocation of storage to those temporaries needing it must be deferred in the code generation phase. However, indirect triples can save some space compared with quadruples if the same temporary value is used more than once. The reason is that two or more entries in the statement array can point to the same line of the op-arg1-arg2 structure.

b)

```
MOV R1,a
MOV R2,b
MUL R2,c
```

ADD R1,R2
SUB R1,d
MOV v,R1

3. Translate the expression a = (a+ b) * (c+ d) + (a + b + c) into
a) Quardruple
b) Triple
c) Indirect Triple
d) 3-address code

Answer **[WBUT 2010, 2011, 2018]**

The quadruples set for the given code is:

	Op	arg1	arg2	result
101	+	a	d	t_1
102	uniminus	t_1		t_2
103	+	c	d	t_3
104	*	t_2	t_3	t_4
105	+	a	b	t_5
106	+	t_5	c	t_6
107	+	t_4	t_6	t_7
108	assign	a	t_7	

The Triples set for the given code is:

	Op	arg1	arg2
101	+	a	d
102	uniminus	(101)	
103	+	c	d
104	*	(102)	(103)
105	+	a	b
106	+	(105)	c
107	+	(104)	(106)
108	assign	a	(107)

For indirect triples, the triples set as above would be accompanied with another level of indirection through a statement table like:

	statement
1	(101)
2	(102)
3	(103)
4	(104)
5	(105)
6	(106)
7	(107)
8	(108)

3-Address code
t1 = a+b
t2 = c+d
t3 = t1 * t2
t4 = - t3
t5 = t1+c
t6 = t4 + t5

4. a) Distinguish between quadruples, triples and indirect triples for the expression. $x = y * -z + y * -z$ **[WBUT 2012]**

Answer:

Quadruple:

	res	op	arg1	arg2
(101)	t1	uniminus	z	
(102)	t2	*	y	t1
(103)	t3	+	t2	t2

Triple:

	op	arg1	arg2
(101)	uniminus	z	
(102)	*	y	(101)
(103)	+	(102)	(102)

Indirect Triple:

	Statement
(1)	(101)
(2)	(102)
(3)	(103)

	op	arg1	arg2
(101)	uniminus	z	
(102)	*	y	(101)
(103)	+	(102)	(102)

b) While the three-address code for the following C program: **[WBUT 2012]**

```
main ( )
    {
        int x = 1;
        int y[20];
        while (x ≤ 20)
         y[x] = 0;
    }
```

Answer:

100 : if (x<=20) goto 102
101 : goto 150
102 : T1=addr(y)
103 : T1[20]=0
104 : goto 100
105 :

5. Generate the three address code for the following code segment (show the semantic actions). **[WBUT 2013]**

```
Main ( )
    {
    int a=1, z=0;
    int b[0];
    while (a<=10)
            b[a]=2*a;
            z=b[a]+a;
    }
```

Answer:

100: if (a<=10) goto 107
101: goto 105
102: T_1 = addr (b)
103: T_2 = 2 * a
104: $T_1[20]=T_2$
105: z = $T_1[20]$ + a
106: goto 100
107:

6. a) Translate the following expression: **[WBUT 2014]**

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

(i) Quadruples
(ii) Triples
(iii) Indirect triples

b) What are the differences among Quadruples, Triples and Indirect Triples?

c) Generate machine code for the following instruction:

$$V = a + (b * c) - d$$

Answer:

a) ***Refer to Question No. 1(a) of Long Answer Type Questions.***
b) ***Refer to Question No. 2(a) of Long Answer Type Questions.***
c) ***Refer to Question No. 2(b) of Long Answer Type Questions.***

7. Generate three-address code for the code fragment **[WBUT 2015]**

```
while(i<10)
{       x=0;
        y=x+2;
        i=i+1;
}
```

and implement it in quadruples, triples and indirect triples.

Answer:

Three address code is as follows:

100 : if i < 10 goto L2
101: goto 106
102: x=0
103: y= x+2
104: i=i+1
105: goto 100

Quadrupie is as follows:

L1: if i>10 goto L2
x=0
y = x+2
t2:= i+1
t3:= t2>10
if t3 goto L1
L2

Triples is as follows:

(1) i>10
(2) if (1), (10)
(3) 0
(4) x st (3)
(5) x+2
(6) y = (5)
(7) i+1
(8) (7) < 10
(9) if (8) , 3

Indirect triples is as follows:

Statement	
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)
(6)	(19)
(7)	(20)
(8)	(21)
(9)	(22)

So from the triples the indirect triples will be

(14) i>10
(15) if (1), (23)
(16) 0
(17) x st (16)
(18) x+2
(19) y = (18)
(20) i+1
(21) (20) < 10
(22) if (21) , 3

8. Translate the following expression:
$m = m + n \times -p + n \times -p$ **into** **[WBUT 2017]**

(i) Quadruples, (ii) Triples, (iii) Indirect triples.

Answer:

$$m = m + n \times -p + n \times -p$$

three-address code:

$$t_1 = p$$
$$t_2 = -t_1$$
$$t_3 = n \times t_2$$
$$t_4 = m + t_3$$
$$m = t_4 + t_3$$

Quadruples:

	OP	Res	arg 1	arg 2
(101)	t_1	–	p	–
(102)	t_2	–	$-t_1$	–
(103)	t_3	*	n	t_2
(104)	t_4	+	m	t_3
(105)	m	+	t_4	t_3

Triples:

	OP	arg 1	arg 2
(101)	–	p	–
(102)	+	$-t_1$	–
(103)	*	n	t_2
(104)	+	m	t_3
(105)	+	t_4	t_3

Indirect triples:

(1)	(101)
(2)	(102)
(3)	(103)
(4)	(104)
(5)	(105)

	OP	arg 1	arg 2
(101)	–	p	–
(102)	–	$-t_1$	–
(103)	*	n	t_2
(104)	+	m	t_3
(105)	+	t_4	t_3

9. Give the properties of intermediate representation. **[WBUT 2019]**

Answer:

The properties of Intermediate representation is:

(i) Ease of generation
(ii) Ease of manipulation
(iii) Procedure Size
(iv) Freedom of expression
(v) Level of abstraction.

- The Intermediate representation must be convenient for semantic analysis phase to produce.
- Must be convenient to translate into real assembly code for all desired target machines.

1. RISC processors execute operations that are rather simple. Examples: load, store, add, shift, branch.

IR should represent abstract load, abstract store, abstract add, etc. –

2. CISC processors execute more complex operations. Examples: multiply-add, add to/from memory.

Simple operations in IR may be "clumped" together during instruction selection to form complex operations.

10. Write short notes on the following:

a) Backpatching **[WBUT 2007, 2014]**

b) Issues in the design of a code generator **[WBUT 2016]**

Answer:

The problem in generating three-address codes in a single pass is that we may not know the labels that control must go to at the time jump statements are generated. So to get around this problem a series of branching statements with the targets of the jumps temporarily left unspecified is generated.

BackPatching is the operation of putting the appropriate target address instead of labels when the proper address of the label is determined.

Backpatching Algorithms perform three types of operations:

- makelist(i). This creates a new list containing only i, an index into the array of quadruples (i.e., the address of a three-address code) and returns pointer to the list it has made.

- merge(i; j). This concatenates the lists pointed at by i and j and returns a pointer to the concatenated list.
- backpatch(p; i). This inserts i as the target address for each of the statements on the list pointed to by p.

An example translation for the rule
$E \rightarrow E^{(1)}$ or $ME^{(2)}$ is
{
backpatch(E1.falselist,M.quad)
E.truelist = merge(E1.truelist,E2.truelist)
E.falselist = E2.falselist}

b) Issues in the design of a code generator

While the details are dependent on the target language and the operating system, issues such as memory management, instruction selection, register allocation, and evaluation order are inherent in almost all code-generation problems. In this section, we shall examine the generic issues in the design of code generators.

Input to the Code Generator

The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the intermediate representation.

Target Programs

The output of the code generator is the target program. Like the intermediate code, this output may take on a variety of forms: absolute machine language, relocatable machine language, or assembly language.

Memory Management

Mapping names in the source program to addresses of data objects in run-time memory is done coorperatively by the front end the code generator.

Instruction Selection

The nature of the instruction set of the target machine determines the difficulty of instruction selection. The uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling.

Register Allocation

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of registers is particularly important in generating good code. The use of registers is often subdivided into two subproblems:

1. During register allocation, we select the set of variables that will reside in registers at a point in the program.
2. During a subsequent register assignment phase, we pick the specific register that a variable will reside in.

CODE OPTIMIZATION

Multiple Choice Type Questions

1. Which of the following is not a loop optimization? [WBUT 2006, 2008, 2009, 2011, 2016, 2018]
   a) Induction variable elimination
   b) Loop unrolling
   c) Loop jamming
   d) Loop heading

Answer: (d)

2. A basic block can be analyzed by [WBUT 2010, 2012, 2015]
   a) DAG
   b) Flow graph
   c) Graph with cycles
   d) None of these

Answer: (a)

3. The method which merges the bodies of two loops is [WBUT 2010, 2017]
   a) loop unrolling b) loop ramming c) constant folding d) none of these

Answer: (b)

4. Optimization(s) connected with $x:=x+0$ is/are [WBUT 2012]
   a) peephole and algebraic
   b) reduction in strength and algebraic
   c) peephole only
   d) loop and peephole

Answer: (c)

5. The graph that shows basic blocks and their successor relationship is called
   a) DAG
   b) Flow chart [WBUT 2013]
   c) Control graph
   d) Hamiltonian graph

Answer: (b)

6. The peep-hole optimization [WBUT 2014]
   a) is applied to a small part of the code
   b) can be used to optimize intermediate code
   c) can be applied to a portion of the code that is not contiguous
   d) all of these

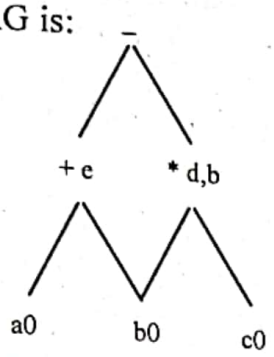
Answer: (a)

Short Answer Type Questions

1. Construct the DAG for the following basic block. [WBUT 2006, 2008, 2009, 2017]

```
d := b * c
e := a + b
b := b * c
a := e - d
```

Answer:
The required DAG is:



2. What is Basic Block? **[WBUT 2013]**

Answer:
Informally, a basic block is a maximal instruction sequence with linear control flow. Only the last instruction of a basic block is allowed to be a jump instruction branching to some other part of the code. The control flow of a program is represented by a graph structure where each mode represents a basic block. Directed edges are inserted between basic blocks B_i and B_j whenever the termination of B_i can directly lead to the execution of B_j.

3. What is code optimization? How is it achieved? **[WBUT 2016]**

Answer:
Refer to Question No. 6(b) of Long Answer Type Questions.

Long Answer Type Questions

1. a) What is Peephole optimization? **[WBUT 2006, 2008, 2009, 2016]**

b) Consider some interblock code optimization without any data-flow analysis by treating each extended basic block as if it is a basic block.
Give algorithms to do the following optimizations within an extended basic block. In each case, indicate what effect on other extended basic blocks a change within one extended basic block can have. **[WBUT 2006, 2008, 2009]**

i. Common subexpression elimination.
ii. Constant folding.
iii. Copy propagation.

Answer:
a) Algorithmic code-generations strategy as in syntax-directed translation, often produce target code that contains redundant instructions and constructs that are not optimal. A simple but effective technique for improving the target code is peephole optimization—a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.
The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous, although some implementations do require this. Program transformations that are characteristic of peephole optimization are:

- Redundant instructions elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Redundant Loads and Stores

In the instructions sequence

MOV R0, a

MOV a, R0

it is possible to delete the second instruction because whenever that is executed, the previous instruction will ensure that the value of a is already in register R0. However, If there had been a label with the second instruction, one could not be sure that it will always executed immediately after the previous one and so may not remove it.

Unreachable Code

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1.In C, the source code might look like:

```
#define debug 0
...
if (debug) {
hspace15ptprint debugging information
}
```

In the intermediate representations the if-statement may be translated as:

```
if debug =1 goto L1
goto L2
L1: print debugging information
L2:
```

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of debug, the code above can be replaced by:

```
if debug != 1 goto L2
print debugging information
L2:
```

As "debug" evaluates to the constant 0, the code becomes:

```
if 0 != 1 goto L2
print debugging information
L2:
```

As the argument of the "if" in the first statement evaluates to a constant 'true', it can be replaced by goto L2. Then all the statement that print debugging information are manifestly unreachable and can be eliminated one at a time.

Flow-of-Control Optimizations

The intermediate codes sometimes generate jumps to jumps or jumps to conditional jumps (or vice versa). Such unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations: We can replace the jump sequence

```
goto L1
...
L1 : gotoL2
```

by the sequence

```
goto L2
...
L1: goto L2
```

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump.

Similarly, the sequence

```
if a < b goto L1
...
L1: goto L2
```

can be replaced by

```
if a < b goto L2
...
L1: goto L2
```

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

```
goto L1
...
L1:if a < b goto L2
L3:
```

may be replaced by

```
L1:if a < b goto L2
goto L3
...
L3:
```

While the number of instructions in the last two examples is the same, we sometimes skip the unconditional jump in the second one, but never in the first one. Thus the second example is superior in execution time.

Algebraic Simplification

A large number of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

x := x + 0
or
x := x * 1
are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization. Common sub-expressions may also be eliminated during peephole optimization. Its possible that a large amount of dead (useless) code may exist in the program. Eliminating these will definitely optimize the code Reduction in strength is also used wherever possible.

Use of Machine Idioms
The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i : = i + 1.

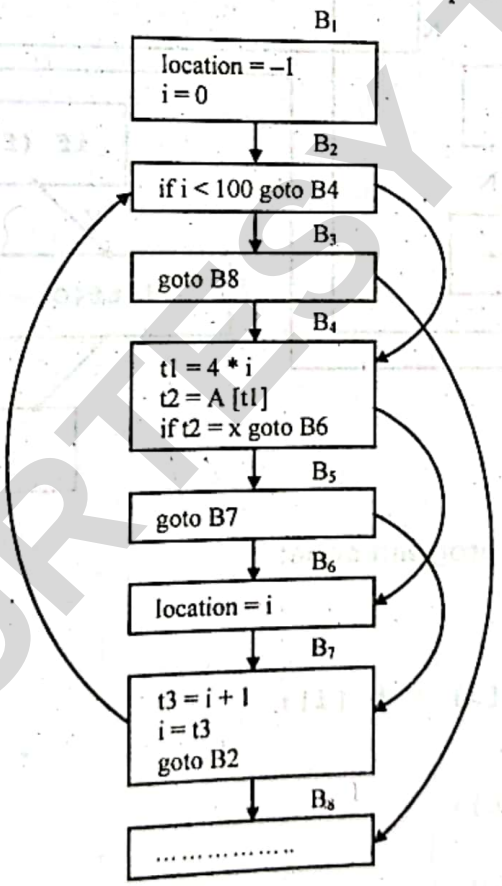
b) A extended basic block is a sequence of blocks B1; B2; : : : ; Bk such that for i _ i < k, Bi is the only predecessor of Bi+1 and B1 does not have a unique predeccor.
We note that an extended basic block is basically a tree of basic blocks rooted at B1 where an edge is a control-flow edge. We start with B1 and its DAG as being the current DAG.
We can "visit" all the basic blocks in the tree in a "depth-first" manner. As we move down a node (say Bi, we modify the current DAG by "adding" the DAG for Bi to the current DAG. On the other hand, if we move up, from Bi we "subtract" the DAG for Bi from the current DAG. This can be achieved if we keep a stack of DAG-s. When we move down to Bi, we merge the DAG of Bi with the current DAG and push it into the stack. Conversely, when we move up, we simply pop out the current DAG from the stack.
We carry out usual method of optimization using the current DAG (by assuming that it is the only DAG of the current basic block) after moving down to a new node Bi by carrying out common subexpression elimination, constant folding and copy propagation only for the codes in Bi. This ensures that no basic block is used more than once for code generation.
In all the cases, a change within one extended basic block will have no effect on other extended basic blocks because no two extended basic block shares any basic block. Hence, changes would remain localized. However, forming data-flow equations may become intractable.

2. Draw the flow graph for the following code: **[WBUT 2010, 2011, 2013]**

i) location = - 1
ii) i = 0
iii) i < 100 goto 5
iv) goto 13
v) t_1 = 4i
vi) t_2 = A [t_1]
vii) if t_2 = x goto 9
viii) goto 10
ix) location = i
x) t_3 = i +1
xi) i = t_3
xii) goto 3
xiii).....

Answer:

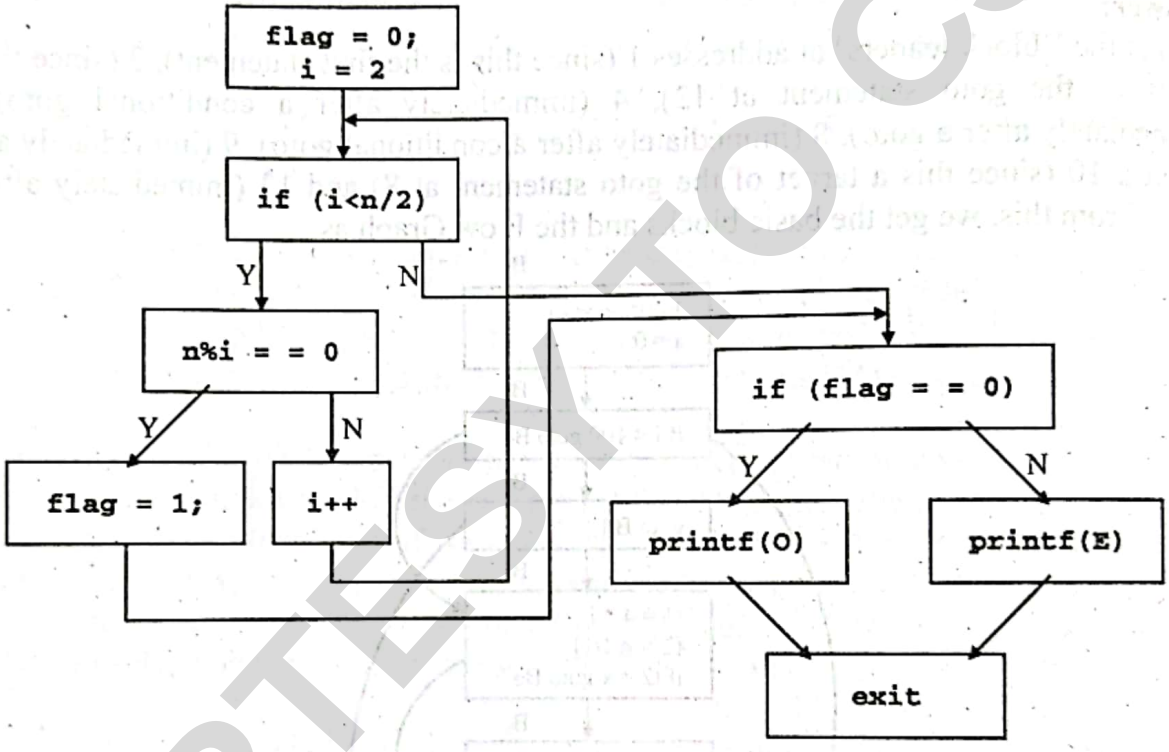
We get the ``block leaders" at addresses 1 (since this is the first statement), 3 (since this a target of the goto statement at 12), 4 (immediately after a conditional goto), 5 (immediately after a goto), 8 (immediately after a conditional goto), 9 (immediately after a goto), 10 (since this a target of the goto statement at 8) and 13 (immediately after a goto). From this, we get the basic blocks and the Flow Graph as:



3. Draw the flow graph for the following code: [WBUT 2018]

```
Check (int n)
        flag = 0;
        for (i = 2; i<n/2; i++) {
        if (n % I = = 0) {
        flag = 1;
        break;
        }
        }
        if (flag = = 0)
        printf("Number is odd");
        else print("Number is even");
        exit
```

Answer:



4. Consider the following program code: [WBUT 2019]

```
Prod = 0, I = 1;
Do
{
Prod = Prod +a [i] * b [i];
 I = i + 1;
}
while (i < = 10);
```

a) Partition in into blocks
b) Construct the flow graph.

Answer:

a) Partition in into blocks:

Input: It contains the sequence of three address statements.

Output: It contains a list of basic blocks with each three address statement in exactly one block.

Method: First identify the leader in the code. The rules for finding leaders are as follows:

- The first statement is a leader.
- Statement L is a leader if there is an conditional or unconditional goto statement like: if....goto L or goto L.
- Instruction L is a leader if it immediately follows a goto or conditional goto statement like: if goto B or goto B.

For each leader, its basic block consists of the leader and all statement up to. It doesn't include the next leader or end of the program.

Consider the following source code for dot product of two vectors a and b of length 10:

The three address code for the above source program is given below:

B1

(1) prod := 0 (2) i := 1

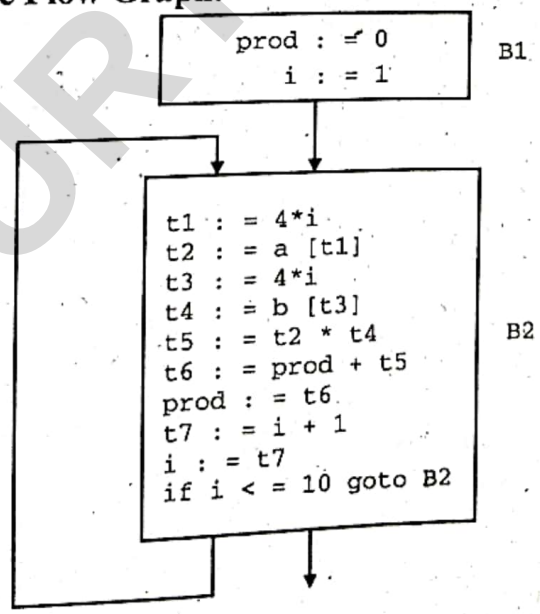
B2

(3) t1 := 4* i
(4) t2 := a[t1]
(5) t3 := 4* i
(6) t4 := b[t3]
(7) t5 := t2*t4
(8) t6 := prod+t5
(9) prod := t6
(10) t7 := i+1
(11) i := t7
(12) if i<=10 goto (3)

Basic block B1 contains the statement (1) to (2).

Basic block B2 contains the statement (3) to (12)

b) Construction of the Flow Graph:



5. Write short note with example to optimize the code: [WBUT 2019]
a) Dead code elimination
b) Variable elimination
c) Code motion
d) Reduction in strength

Answer:

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. Compiler optimizing process should meet the following objectives:

- The optimization must be correct, it must not, in any way, change the meaning 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.

Example:

Code Optimization is done in the following different ways:

a) Dead code elimination: Variable propagation often leads to making assignment statement into dead code.

```
int global;
void f ()
{
  int i;
  i = 1;          /* dead store */
  global = 1;     /* dead store */
  global = 2;
  return;
  global = 3;     /* unreachable */
}
```

Below is the code fragment after dead code elimination.

```
int global;
void f ()
{
  global = 2;
  return;
}
```

b) Variable elimination:

```
int a[SIZE];
int b[SIZE];

void f (void)
{
  int i1;

  for (i1 = 0; i1 < SIZE; i1++)
```

```
    a[i1] = b[i1];
  return;
}
```

Induction variable elimination can reduce the number of additions (or subtractions) in a loop, and improve both run-time performance and code space. Some architectures have auto-increment and auto-decrement instructions that can sometimes be used instead of induction variable elimination.

c) Code motion:

- Reduce the evaluation frequency of expression.
- Bring loop invariant statements out of the loop.

```
a = 200;
 while(a>0)
 {
     b = x + y;
     if (a % b == 0}
     printf("%d", a);
   }

//This code can be further optimized as
a = 200;
b = x + y;
while(a>0)
 {
     if (a % b == 0}
     printf("%d", a);
   }
```

d) Reduction in Strength:

- An induction variable is used in loop for the following kind of assignment i = i + constant.
- Strength reduction means replacing the high strength operator by the low strength.

```
i = 1;
while (i<10)
{
    y = i * 4;
}
//After Reduction
i = 1
t = 4
{
   while( t<40)
   y = t;
   t = t + 4;
}
```

6. Write short note on the following:
a) Peephole Optimization **[WBUT 2007, 2010, 2011, 2013, 2015]**
b) Code optimization **[WBUT 2008]**
c) Loop optimization **[WBUT 2009]**
d) Constant folding and copy propagation **[WBUT 2014]**

Answer:

a) Peephole Optimization:

Refer to Question No. 1(a) of Long Answer Type Questions.

b) Code optimization:

In the context of a program written in a high level language and then compiled by a suitable compiler, the final machine executable code, leaves enough scope for optimization. There can be two dimensions to the optimization:

- We may try to optimize the code such that it runs faster.
- We may want the compiled code to occupy as less memory as possible. In many cases, reduction in size amounts to reduction in run-time (since less number of instructions gets executed) but quite often, the two dimensions are often antagonistic to each other.

There can be several levels at which the code can be optimized:

Design Level: Here the designer (human being) selects the best possible algorithms and tries to implement that by writing good quality code.

Source Code Level: Here the programmer (human being) tries to achieve best possible coding quality by avoiding obvious slowdowns.

Compile Level: Here, an optimizing compiler tries to ensure that the executable program is optimized at least as much as the compiler can predict.

Assembly Level: A human programmer writing code using an assembly language designed for a particular hardware platform will normally produce the most efficient code since the programmer can take advantage of the full repertoire of machine instructions.

Compiler level code optimization involves the application of rules and algorithms (by the compiler) on the generated code at different level to transform the same to a code that is faster, smaller, more efficient, and so on. Code optimization is therefore the process of transformation of the generated non-optimal code to an optimal one.

c) Loop optimization:

Loops, especially the inner loops are where programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if that increases the amount of code outside that loop. Three techniques are important for loop optimization:

- Code Motion. Moving "loop invariant" code outside a loop.
- Induction-variable Elimination. Eliminating all but one variable that change in step within a loop.
- Reduction in Strength. Replacing an expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion

An expression that yields the same result independent of the number of times a loop is executed is called a loop-invariant computation.

In code motion, the transformation takes a loop-invariant computation and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of 'limit – 2' is a loop-invariant computation in the following while-statement:

```
while (i <= limit - 2)
```

Code motion will result in the equivalent of

```
t = limit - 2;
while (i <= t)
```

Induction Variable Elimination and Reduction in Strength

An induction variable is a variable that gets increased or decreased by a fixed amount on every iteration of a loop, or is a linear function of another induction variable. For example, in the following loop, i and j are induction variables:

```
for (i = 0; i < 10; ++i) {
        j = 17 * i;
}
```

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. Strength reduction is an optimization technique where a costly operation is replaced with an equivalent but less expensive operation. Some examples are (assume 'j' is an integer and 'x' is a floating point variable):

- "j * 16" can be replaced by "j << 4"
- "j / 16" can be replaced by "j >> 4"
- "j * 2" can be replaced by "j + j"
- "2.0 * x" can be replaced by "x + x"
- Even "j * 15" can be replaced by "(j << 4) - j"

Induction variable elimination quite often goes hand in hand with strength reduction.

d) Constant folding and copy propagation:

Constant folding and constant propagation are related compiler optimizations used by many modern compilers. An advanced form of constant propagation known as sparse conditional constant propagation can more accurately propagate constants and simultaneously remove dead code.

Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime. Terms in constant expressions are typically simple literals, such as the integer literal 2, but they may also be variables whose values are known at compile time. Consider the statement:

```
i = 320 * 200 * 32;
```

Most modern compilers would not actually generate two multiply instructions and a store for this statement. Instead, they identify constructs such as these and substitute the

computed values at compile time (in this case, 2,048,000). The resulting code would load the computed value and store it rather than loading and multiplying several values.
Constant folding can even use arithmetic identities. The value of 0 * x is zero even if the compiler does not know the value of x.
Constant folding may apply to more than just numbers. Concatenation of string literals and constant strings can be constant folded. Code such as "abc" + "def" may be replaced with "abcdef".
Constant folding can be done in a compiler's front end on the IR tree that represents the high-level source language, before it is translated into three-address code, or in the back end, as an adjunct to constant propagation.
Constant propagation is the process of substituting the values of known constants in expressions at compile time. Such constants include those defined above, as well as intrinsic functions applied to constant values.
Consider the following pseudocode:

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

Propagating x yields:

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

Continuing to propagate yields the following (which would likely be further optimized by dead code elimination of both x and y.)

```
int x = 14;
int y = 0;
return 0;
```

Constant propagation is implemented in compilers using reaching definition analysis results. If all a variable's reaching definitions are the same assignment, which assigns a same constant to the variable, then the variable has a constant value and can be replaced with the constant.
Constant propagation can also cause conditional branches to simplify to one or more unconditional statements, when the conditional expression can be evaluated to true or false at compile time to determine the only possible outcome.

QUESTION 2015

Group – A
(Multiple Choice Type Questions)

1. Choose the correct alternatives for the following:

i) Role of preprocessor is to

a) produce output data
b) produce output to compilers
✓c) produce input to compilers
d) none of these

ii) A Top-Down Parser generates

✓a) left most derivation
b) right most derivation in reverse
c) right most derivation
d) none of these

iii) Type checking is done normally during

a) lexical analysis
b) syntax analysis
✓c) syntax directed translation
d) code generation

iv) The language produced by the regular grammar $S \rightarrow aS|bS|a|b$ is

a) a^*b^*
b) aa^*bb^*
c) $(a+b)^*$
✓d) $(a+b)(a+b)^*$

v) The grammar $S \rightarrow S\alpha_1|S\alpha_2|\beta_1|\beta_2$

✓a) is left recursive
b) has common left factor
c) is left recursive and also has common left factor
d) is a CFG

vi) A basic block can be analyzed by a

✓a) DAG
b) Flow graph
c) Graph with cycles
d) None of these

vii) Which of the following is not an intermediate code form?

a) Quadruples
b) Triples
✓c) Abstract syntax tree
d) Indirect triples

viii) An inherited attribute is one whose initial value at a parse tree node is defined in terms of
a) attribute values of its children only
b) attribute values of itself and its children
✓c) attribute values of its parents and/or its siblings
d) none of these

ix) Consider the program statement $b = 2$ where b is a Boolean variable. Which stage of compilation can detect this error?
a) Lexical analysis
b) Syntax analysis
✓c) Semantic analysis
d) Code generation

x) Given the grammar $S \to ABc,\ A \to a|\varepsilon,\ B \to b|\varepsilon$ FOLLOW(A) is the set
a) $\{\$\}$
b) $\{b\}$
✓c) $\{b, c\}$
d) $\{a, b, c\}$

Group – B
(Short Answer Type Questions)

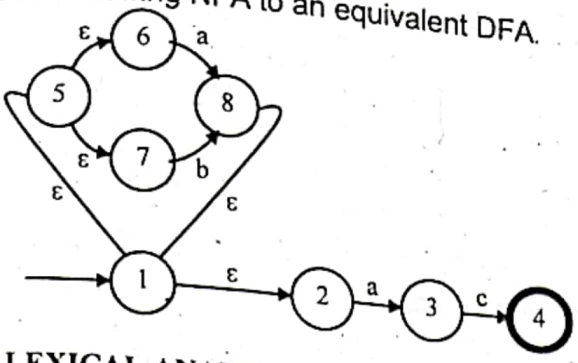
2. Explain inherited attribute and synthesized attribute for syntax directed translation with suitable example.
See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 3.

3. What is activation record? Explain clearly the components of activation record.
See Topic: RUNTIME ENVIRONMENT, Short Answer Type Question No. 1.

4. Generate an annotated parse tree for the string "3 + 2 – 4" using the grammar
$$E \to E + T | E - T | T \quad T \to 0|1|2|\ldots.|9$$
See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 9.

5. Describe the algorithm for eliminating left recursion from a CFG. Eliminate left recursion from the following grammar.
$$S \to Aa|b$$
$$A \to Ac|Sd|\varepsilon$$
See Topic: OPERATOR PRECEDENCE PARSING, Short Answer Type Question No. 2.

6. Convert the following NFA to an equivalent DFA.



See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 9.

Group – C
(Long Answer Type Questions)

7. a) How the following statement is translated via the different phases of compilation? Explain.

MOTION = DISTANCE + RATE * DISPLACEMENT + 70.

b) What is an operator precedence parser? List the advantages and disadvantages of operator precedence parsing.

c) What do you mean by Thomson Construction? Explain with an example.

a) See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 3.

b) See Topic: OPERATOR PRECEDENCE PARSING, Long Answer Type Question No. 1(b).

c) See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 1.

8. Consider the following grammar:

$E \rightarrow E + T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

(i) Obtain the FIRST and FOLLOW sets for the above grammar.

(ii) Construct the predictive parsing table for the above grammar.

See Topic: TOP-DOWN PARSING, Long Answer Type Question No. 1.

9. Consider the grammar $G = \{V, T, S, P\}$; where $V = \{S, A\}, T = \{a, b\}$, S is the start variable and $P = \{S \rightarrow AS|b,\ A \rightarrow SA|a\}$.

(i) Compute the collection of sets of LR(0) item sets for the grammar.

(ii) Construct the SLR parsing table using the SLR algorithm.

(iii) Show all moves allowed by the table from (ii) on the input $abab$.

See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Long Answer Type Question No. 6.

10. a) Generate three-address code for the code fragment

```
while(i<10)
{       x=0;
        y=x+2;
        i=i+1;
}
```

and implement it in quadruples, triples and indirect triples.

b) Translate the arithmetic expression $a*-\left(b+\frac{c}{d}\right)$ into syntax tree and postfix notation.

c) Assuming 3 registers available generate machine code for the instruction $X=\frac{a}{-(b*c)}-d$.

a) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 7.
b) See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 2.
c) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 1(c).

11. Write short notes on any three of the following:
   a) Dependency Graph
   b) L-attribute definition
   c) Peephole optimization
   d) Left factoring
   e) Symbol table

a) See Topic: SYNTAX DIRECTED TRANSLATION, Long Answer Type Question No. 5(b).
b) See Topic: SYNTAX DIRECTED TRANSLATION, Long Answer Type Question No. 5(d).
c) See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 6(a).
d) See Topic: PARSING AND CONTEXT FREE GRAMMAR, Long Answer Type Question No. 5(a).
e) See Topic: RUNTIME ENVIRONMENT, Long Answer Type Question No. 4(a).

QUESTION 2016

Group – A
(Multiple Choice Type Questions)

1. Choose the correct alternatives for any *ten* of the following:

i) Which data structure is mainly used during shift-reduce parsing?

✓a) Stack
b) Array
c) Queue
d) Pointer

ii) If x is a terminal, then FIRST (x) is

a) null

✓c) {x}

b) x*

d) none of these

iii) The edge in a flow graph whose heads dominate their tails are called

✓a) Back edge

c) Front edges

b) Flow edges

d) None of these

iv) Which of the following is not a loop optimizer?

a) Loop unrolling

✓c) Loop heading

b) Loop jamming

d) Induction variable elimination

v) The regular expression $(a|b)^* abb$ denotes

a) all possible combination of a's and b's

✓b) set of all strings ending with abb

c) set of all strings starting with a and ending with abb

d) none of these

vi) Shift reduce parsers are

a) top-down parsers

c) may be top-down or bottom up parsers

✓b) bottom up parsers

d) none of these

vii) The following productions of a regular grammar generates a language L.

$S \rightarrow aS|bS|a|b$

The regular expression for L is

a) $a+b$

✓c) $(a+b)^*$

b) $(a+b)(a+B)^*$

d) $(aa+bb)a^*$

viii) The regular expression representing the set of all strings over *(x, y)* ending with *xx* beginning with *y* is

a) $xx(x+y)^* y$

c) $yy(x+y)^* x$

✓b) $y(x+y)^* xx$

d) $y(xy)^* xx$

ix) The basic limitation of Finite State Machine is that

✓a) it cannot remember arbitrary large amount of information

b) it cannot recognize grammars that are regular

c) it sometimes recognize grammars that are not regular

d) all of these

x) An annotated parse tree is a parse tree

a) with values of only some attributes shown at parse tree nodes

✓b) with attribute values shown at the parse node

c) without attribute values shown at the parse tree nodes

d) with grammar symbols at the parse tree nodes

xi) Which one of the following errors will not be denoted by the compiler?

a) Lexical error

b) Semantic error

c) Syntactic error

✓d) Logical error

xii) If a grammar is in LALR (1) then it is necessarily

a) LL(1)

✓b) LR(1)

c) SLR(1)

d) none of these

Group – B

(Short Answer Type Questions)

2. Find out regular expression corresponding to the finite automata:

PS	Next State	Next State
	a	b
q1	q1, q2	–
q2	q3	q2, q2
q3	q2	–

See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 10.

3. Translate the arithmetic expression $a*-(b+c)$ into:

a) Syntax tree

b) Three-address code

c) Postfix notation

a) See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 2.

b) & c) See Topic: INTERMEDIATE CODE GENERATION, Short Answer Type Question No. 1.

4. What is handle?

Consider the grammar $E \rightarrow E+E \mid E*E \mid id$

Find the handle of the right sentential forms of reduction for the string id + id + id.

See Topic: PARSING AND CONTEXT FREE GRAMMAR, Short Answer Type Question No. 4.

5. What is type checking? Differentiate between Dynamic and Static type checking.

See Topic: TYPE CHECKING, Short Answer Type Question No. 1.

6. What is look ahead operator? Give an example with the help of the look ahead concept. Show how identifiers can be distinguished from keywords.

See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 2.

Group – C
(Long Answer Type Questions)

7. a) What is symbol table? How is it implemented?

b) Explain Syntax and Semantic error.

c) What is code optimization? How is it achieved?

a) & b) See Topic: RUNTIME ENVIRONMENT, Long Answer Type Question No. 2(a) & (b).

c) See Topic: CODE OPTIMIZATION, Short Answer Type Question No. 3.

8. Explain the following terms with the example given below:

$a := b*(c+d/b)-(e*f)$

a) Quadruples

b) Triples

c) Indirected Triples

See Topic: INTERMEDIATE CODE GENERATION, Short Answer Type Question No. 2.

9. a) Consider the following grammar and design a SLR parser table:

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

b) Make a comparison between Predictive Parser and Shift Reduce Parser.

c) What is Ambiguous Grammar?

a) & b) See Topic: TOP-DOWN PARSING, Long Answer Type Question No. 6(a) & (b).

c) See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 7.

10. a) What is peephole optimization?

b) What is an activation record? When and why are those records used?

c) Generate three address code for the following program segment:

```
while (a < c and b > d) do
if a = 1 then c = c + 1;
else
while a < = d do
a = a+ 3;
```

a) See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 1(a).

b) See Topic: RUNTIME ENVIRONMENT, Long Answer Type Question No. 1(1st & 2nd Part).

c) See Topic: INTERMEDIATE CODE GENERATION, Short Answer Type Question No. 4.

11. Write the short notes any *three* of the following:

a) Symbol table organization

b) YACC

c) Issues in the design of a code generator

d) Cross compiler

e) Context-free grammar

a) See Topic: RUNTIME ENVIRONMENT, Long Answer Type Question No. 4(a).

b) See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 3(a).

c) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 10(b).

d) See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 5(a).

e) See Topic: PARSING AND CONTEXT FREE GRAMMAR, Long Answer Type Question No. 5(b).

QUESTION 2017

Group – A

(Multiple Choice Type Questions)

1. Choose the correct alternatives for the following:

i) Which of the following is an example of bottom up parsing?

a) LL parsing

b) Predictive Parsing

c) Recursive descent parsing

✓d) Shift-reduce parsing

ii) The output of the parser is

a) tokens

b) syntax tree

✓c) parse tree

d) non-terminals

iii) If x is a terminal, then FIRST (x) is

a) ε

✓b) $\{x\}$

c) $x*$

d) none of these

iv) YACC builds up

✓a) SLR parsing table

b) LALR parsing table

c) canonical LR parsing table

d) none of these

v) Which one of the following is a top-down parser?

✓a) Recursive descent parser

b) Operator precedence parser

c) An LR(k) parser

d) An LALR(k) parser

vi) Cross-compiler is a compiler

a) which is written in a language that is different from the source language

b) that generates object code for host machine

c) which is written in a language that is same as the source language

✓d) that runs on one machine but produces object code for another machine

vii) An ideal compiler should

a) be smaller in size

b) take less time for compilation

c) be written in a high level language

✓d) produce object code that is smaller in size and executes faster

viii) Synthesized attribute can easily be simulated by an

a) LL grammar

✓b) LR grammar

c) ambiguous grammar

d) none of these

ix) Type checking is normally done during

a) lexical analysis

b) syntax analysis

✓c) syntax directed translation

d) code generation

x) The method which merges the bodies of two loops is

a) loop rolling

✓b) loop jamming

c) constant folding

d) none of these

Group – B

(Short Answer Type Questions)

2. Consider the context-free grammar:

$S \rightarrow SS+|SS*|\varepsilon$

a) How the string $aa+a*$ can be generated by this grammar?

b) Construct a parse tree for this string.

See Topic: PARSING AND CONTEXT FREE GRAMMAR, Short Answer Type Question No. 8.

3. What is handle? Consider the grammar $E \to E + E | E * E | id$

Find the handles of the right sentential forms of the reduction for the string $id + id * id$.

See Topic: PARSING AND CONTEXT FREE GRAMMAR, Short Answer Type Question No. 4.

4. Eliminate the left-recursion for the following grammar:

$S \to (L) | a$

$L \to LS | S$

See Topic: PARSING AND CONTEXT FREE GRAMMAR, Short Answer Type Question No. 9.

5. Construct the DAG for the following basic block:

$d = b * c$

$e = a + b$

$b = b * c$

$a = e - d$

See Topic: CODE OPTIMIZATION, Short Answer Type Question No. 1.

6. What is recursive descent parsing? Describe the drawbacks of recursive descent parsing for generating the string 'abc' from the grammar:

$S \to aBc$

$B \to bc | b$

See Topic: TOP-DOWN PARSING, Short Answer Type Question No. 3.

Group – C
(Long Answer Type Questions)

7. a) Construct NFA from the regular expression using Thompson's method $L = aa(a|b)^* ab$.

b) Write regular definition for the following language:

All strings of letter that contain the five vowels in order.

c) Construct the predictive parsing table for the following grammars:

$S \to AaAb | BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

See Topic: PARSING AND CONTEXT FREE GRAMMAR, Long Answer Type Question No. 4.

8. Construct the LR(0) set of items and the SLR parsing table for the following grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Long Answer Type Question No. 7.

9. a) Translate the following expression:

$m = m + n \times -p + n \times -p$ into.

(i) Quadruples, (ii) Triples, (iii) Indirect triples.

b) What are the differences among Quadruples, Triples and Indirect triples?

c) Design a direct acyclic graph for the string:

$a + a*(b-c) + (b-c)*d$.

a) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 8.

b) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 2(a).

c) See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 11(a).

10. a) Consider the following grammar:

$S \rightarrow CC$

$C \rightarrow cC \mid d$

Find the LR(1) set of items.

b) Describe about the operator precedence parser.

c) What are the two types of attributes that are associated with a grammar symbol?

a) See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Long Answer Type Question No. 8.

b) See Topic: OPERATOR PRECEDENCE PARSING, Short Answer Type Question No. 3.

c) See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 11(b).

QUESTION 2018

Group – A

(Multiple Choice Type Questions)

1. Choose the correct alternatives of the following:

i) What is the output of lexical analyzer?

a) A parse tree

✓b) A list of tokens

c) A syntax tree

d) None of these

ii) Parse tree is generated in the phase of

✓a) Syntax Analysis
b) Semantic Analysis
c) Code Optimization
d) Intermediate Code Generation

iii) Shift reduce parsers are

a) top down parser
b) may be top down or bottom up
✓c) bottom up parser
d) None of these

iv) The grammar S → aSa | bS | c is

a) LL(1) but not LR (1)
b) LR(1) but not LL(1)
✓c) Both LL(1) and LR(1)
d) None of these

v) White spaces and Tabs are removed in

✓a) Lexical Analysis
b) Syntax Analysis
c) Semantic Analysis
d) All of these

vi) Left factoring guarantees

a) not occurring of backtracking
✓b) cycle free parse tree
c) error free target code
d) correct LL(1) parsing table

vii) A parse tree showing the values of attributes at each node is called in particular

a) syntax tree
✓b) annotated parse tree
c) syntax direct parse tree
d) direct acyclic graph

viii) Which of the following is not true for Dynamic Type Checking?

a) It increases the cost of execution
b) Type checking is done during the execution
c) All the type errors are detected
✓d) None of the above

ix) Which of the following is not a loop optimization?

a) Induction variable elimination
b) Loop jamming
c) Loop unrolling
✓d) Loop heading

x) YACC builds up

a) SLR parsing table
✓b) LALR parsing table
c) Canonical LR parsing table
d) None of these

Group – B

(Short Answer Type Questions)

2. Describe analysis phase of a Compiler with a block diagram.

See Topic: INTRODUCTION TO COMPILER, Short Answer Type Question No. 1.

3. Describe with diagram the working process of Lexical Analyzer.

See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 5.

4. What is error handling? Describe the Panic Mode and Phrase level error recovery technique with example.

See Topic: PARSING AND CONTEXT FREE GRAMMAR, Short Answer Type Question No. 5.

5. What is ambiguity in grammar? Justify whether the grammar is ambiguous or not. A→AA|(A)|a

See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 12.

6. What is recursive descent parsing? Describe the drawbacks of recursive descent parsing for generating the string 'abc' from the grammar.

S → aBc

B → bc|b

See Topic: TOP-DOWN PARSING, Short Answer Type Question No. 3.

Group – C

(Long Answer Type Questions)

7. Describe with a block diagram the parsing technique of LL(1) parser. Parse the string 'abba' using LL(1) parser where the parsing table is given below.

	A	B	$
S	S → aBa		
B	B → ε	B → bB	

Check whether the following grammar is LL(1) or not:

X → Yz | a

Y → bZ | ε

Z → ε

1st & 2nd Part: See Topic: TOP-DOWN PARSING, Long Answer Type Question No. 4 (a) & (b).

3rd Part: See Topic: TOP-DOWN PARSING, Long Answer Type Question No. 7.

8. Describe LR parsing with block diagram. What are the main advantages of LR parsing? Construct SLR parsing table for the grammar given below.

S → Ab

A → bA/a

1st & 2nd Part: See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Long Answer Type Question No. 4 (a) & (b).
3rd Part: See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Short Answer Type Question No. 2.

9. a) Construct DFA directly from the regular expression:
L = (a | b)*ab
b) What are the main contributions of Syntax Directed Translation in Compiler?
c) Mention different loop optimization techniques. Optimize the following code:

```
do{
        item = 10;
        x = x + item;
}while (value<50);
```

a) See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 4.
b) See Topic: SYNTAX DIRECTED TRANSLATION, Long Answer Type Question No. 4(a).
c) See Topic: SYNTAX DIRECTED TRANSLATION, Long Answer Type Question No. 4(b).

10. a) Translate the expression a = (a+b)*(c+d)+(a+b+c) into
    i) Quadruple
    ii) Triple
    iii) Indirect Triple
b) Draw the flow graph for the following code:

```
Check (int n)
        flag = 0;
        for (i = 2; i<n/2; i++) {
        if (n % I = = 0) {
        flag = 1;
        break;
        }
        }
        if (flag = = 0)
        printf("Number is odd");
        else print("Number is even");
        exit
```

a) See Topic; INTERMEDIATE CODE GENERATION, Long Answer type Question No. 3.
b) See Topic: CODE OPTIMIZATION, Long Answer type Question No. 3.

11. Write the short notes on any *three* of the following:

a) LEX and YAAC
b) Activation Record
c) Symbol table
d) Left Recursion
e) LALR

a) See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 3(d).
b) See Topic: RUNTIME ENVIRONMENT, Long Answer Type Question No. 4(b)
c) See Topic: RUNTIME ENVIRONMENT, Long Answer Type Question No. 4(a).
d) See Topic: OPERATOR PRECEDENCE PARSING, Long Answer Type Question No. 2.
e) See Topic: SYNTAX DIRECTED TRANSLATION, Long Answer Type Question No. 5(d).

QUESTION 2019

Group – A
(Multiple Choice Type Questions)

1. Choose the correct alternatives of the following:

i) ..... or scanning is the process where the stream of characters making up the source program is read from left to right grouped into tokens.

✓a) Lexical Analysis
b) Diversion
c) Modeling
d) None of these

ii) A given grammar is not LL(1) if the parsing table of a grammar may contain

a) any blank field
b) any e-entry
c) duplicate entry of same production
✓d) more than one production rule.

iii) Regular expression (x/y) (x/y) denotes the set

a) {xy, xy}
✓b) {xx, xy, yx, yy}
c) {x, y}
d) {x, y, xy}

iv) The regular expressions denote zero or more instances of an x or y is

a) (x + y)
✓b) (x + y)*
c) (x* + y)
d) (xy)*

v) YACC builds up

a) SLR parsing table
✓b) LALR parsing table
c) Canonical LR parsing table
d) none of these

vi) Grammar of the programing is checked at..... phase of compiler.

a) Semantic analysis
✓b) Syntax analysis
c) Code optimization
d) Code generation

vii) A grammar that produce more than one parse tree

✓a) Ambiguous
b) Unambiguous
c) Regular
d) All of these

viii) Compiler can check ..... Error.

a) logical
✓b) syntax
c) content
d) both (a) and (b)

ix) In operator precedence parsing, precedence relations are defined

a) for all pair of non-terminals
b) for all pair of terminals
c) to delimit the handle
✓d) only for a certain pair of terminals

x) ..... is the most general phase structured grammar.

✓a) Context sensitive
b) Regular
c) Context tree
d) All of these

xi) Inherited attribute is a natural choice in

a) keeping track of variable declaration
b) checking for the correct use of L values and R values
✓c) both (a) and (b)
d) none of these

xii) Number of states of FSM required to simulate behaviour of a computer with a memory capable of storing m words, each of length n

a) m × 2n
✓b) 2mn
c) 2(m + n)
d) all of these

Group – B

(Short Answer Type Questions)

2. Write short notes on symbol table manager.

See Topic: RUNTIME ENVIRONMENT, Long Answer Type Question No. 4(a).

3. Describe the errors encountered in different phases of compiler.

See Topic: INTRODUCTION TO COMPILER, Short Answer Type Question No. 1.

4. Differentiate between top-down and bottom-up approach.

See Topic: TOP-DOWN PARSING, Short Answer Type Question No. 4.

5. Remove left recursion $S \rightarrow Aa/b$, $A \rightarrow Ac/Sd/e$.

See Topic: OPERATOR PRECEDENCE PARSING, Short Answer Type Question No. 4.

6. Define NFA.

See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 11.

Group – C
(Long Answer Type Questions)

7. Consider the following program code:

```
Prod = 0, I = 1;
Do
{
Prod = Prod +a [i] * b [i];
I = i + 1;
}
while (i < = 10);
```

a) Partition in into blocks

b) Construct the flow graph.

See Topic: CODE OPTIMIZATION, Long Answer type Question No. 4.

8. Write short note with example to optimize the code:

a) Dead code elimination

b) Variable elimination

c) Code motion

d) Reduction in strength

See Topic: CODE OPTIMIZATION, Long Answer type Question No. 5.

9. a) Give the properties of intermediate representation.

b) Discuss the concepts of parameter pass mechanisms.

a) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 9.

b) See Topic: RUNTIME ENVIRONMENT, Long Answer Type Question No. 3.

10. Explain in detail the process of compilation. Develop the output of each phase of the compilation for the input a = (b + c) * (b + c) *2

See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 4.

11. What is Regular Expression? Write the regular expression for:

a) R = R1 + R2 (Union operation)

b) R = R1.R2 (Concatenation operation)

c) R = R1* (Kleen Clouser)

d) R = R + (Positive Clouser)

e) Write a regular expression for a language containing strings which end with "abb" over $\sum$ = {a, b}.

f) Construct a regular expression for the language containing all strings having any number of a's and b's except the full string.

See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 2.