CAT 1 COMPILER QB SOLUTIONS

1. a) Explain different types of translators with example.

There are 3 different types of translators as follows:

Compiler →

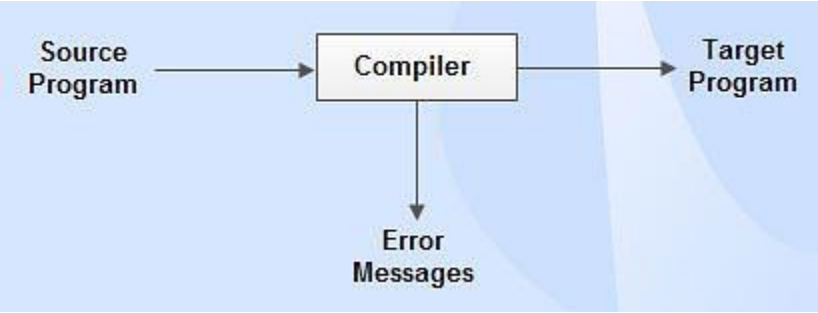
- A compiler is a translator used to convert high-level programming language to low-level programming language.
- It converts the whole program in one session and reports errors detected after the conversion.
- The compiler takes time to do its work as it translates high-level code to lower-level code all at once and then saves it to memory.
- A compiler is processor-dependent and platform-dependent.
- But it has been addressed by a special compiler, a cross-compiler and a source-to-source compiler.

Advantages of the Compiler:

- The whole program is validated so there are no system errors.
- The executable file is enhanced by the compiler, so it runs faster.
- User do not have to run the program on the same machine it was created.

Disadvantages of the Compiler:

- It is slow to execute as you have to finish the whole program.
- It is not easy to debug as errors are shown at the end of the execution.
- Hardware specific, it works on specific machine language and architecture.



Interpreter →

- Interpreter is a translator which is used to convert programs in high-level language to low-level language. Interpreter translates line by line and reports the error once it encountered during the translation process.
- It directly executes the operations specified in the source program when the input is given by the user.
- It gives better error diagnostics than a compiler.

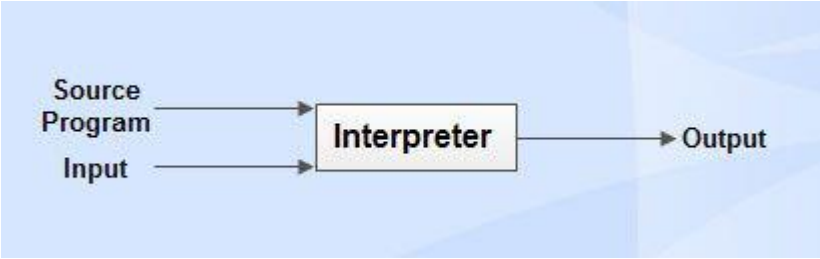
- An interpreter is faster than a compiler as it immediately executes the code upon reading the code.
- An interpreter is also more portable than a compiler as it is not processor-dependent, you can work between hardware architectures.

Advantages of the Interpreter:

- You discover errors before you complete the program, so you learn from your mistakes.
- Program can be run before it is completed so you get partial results immediately.
- You can work on small parts of the program and link them later into a whole program.

Disadvantages of the Interpreter:

- There's a possibility of syntax errors on unverified scripts.
- Program is not enhanced and may encounter data errors.
- It may be slow because of the interpretation in every execution.



Assembler →

- An assembler is a translator used to translate assembly language to machine language.
- It is like a compiler for the assembly language but interactive like an interpreter.
- Assembly language is difficult to understand as it is a low-level programming language.
- An assembler translates a low-level language, an assembly language to an even lower-level language, which is the machine code.
- The machine code can be directly understood by the CPU.

Advantages of the Assembler:

- The symbolic programming is easier to understand thus time-saving for the programmer.
- It is easier to fix errors and alter program instructions.
- Efficiency in execution just like machine level language.

Disadvantages of the Assembler:

- It is machine dependent, cannot be used in other architecture.

- A small change in design can invalidate the whole program.
- It is difficult to maintain.

SI. No	Compiler	Interpreter
1	Performs the translation of a program as a whole.	Performs statement by statement translation.
2	Execution is faster.	Execution is slower.
3	Requires more memory as linking is needed for the generated intermediate object code.	Memory usage is efficient as no intermediate object code is generated.
4	Debugging is hard as the error messages are generated after scanning the entire program only.	It stops translation when the first error is met. Hence, debugging is easy.
5	Programming languages like C, C++ uses compilers.	Programming languages like Python, BASIC, and Ruby uses interpreters.

1. b) Explain various phases of compilers comes under front end .

Lexical Analysis:

Lexical analyzer phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens. It read and breakdown source code into smaller unit called tockens.

The primary functions of this phase are:

- Identify the lexical units in a source code
- Classify lexical units into classes like constants, reserved words, and enter them in different tables. It will Ignore comments in the source program
- Identify token which is not a part of the language

Syntax Analysis :

Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not. It is also called as parser.

Here, is a list of tasks performed in this phase:

- Obtain tokens from the lexical analyzer
- Checks if the expression is syntactically correct or not
- Report all syntax errors
- Construct a hierarchical structure which is known as a parse tree

Semantic Analysis :

Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax. It generally checks the logic or meaning of source code, it checks for semantic errors.

Functions of Semantic analyses phase are:

- Helps you to store type information gathered and save it in symbol table or syntax tree
- Allows you to perform type checking
- In the case of type mismatch, where there are no exact type correction rules which satisfy the desired operation a semantic error is shown
- Collects type information and checks for type compatibility
- Checks if the source language permits the operands or not

Intermediate Code Generation :

In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

Functions on Intermediate Code generation:

- It should be generated from the semantic representation of the source program
- Holds the values computed during the process of translation
- Helps you to translate the intermediate code into target language
- Allows you to maintain precedence ordering of the source language
- It holds the correct number of operands of the instruction

2. a) Explain the following terms
   i) Cross Compilers
   ii) Bootstrap Compilers
   iii) Just in time Compiler

1. Cross Compiler :

1. A compiler that runs on platform (A) and is capable of generating executable code for platform (B) is called a cross-compiler.
2. A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
3. For example, a cross compiler executes on machine X and produces machine code for machine Y.
4. For microcontrollers, we use cross compiler because it doesn't support an operating system.
5. It is useful for embedded computers which are with limited computing resources.
6. To compile for a platform where it is not practical to do the compiling, a cross-compiler is used.
7. It helps to keep the target environment separate from the built environment.

2. Bootstrap Compilers :

1. It is an approach for making a self-compiling compiler that is a compiler written in the source programming language that it determine to compile.
2. A bootstrap compiler can compile the compiler and thus you can use this compiled compiler to compile everything else and the future versions of itself.
3. **Uses of Bootstrapping**

   There are various uses of bootstrapping which are as follows −

   - It can allow new programming languages and compilers to be developed starting from actual ones.
   - It allows new features to be combined with a programming language and its compiler.
   - It also allows new optimizations to be added to compilers.
   - It allows languages and compilers to be transferred between processors with different instruction sets
4. **Advantages of Bootstrapping**

   There are various advantages of bootstrapping which are as follows −

   - Compiler development can be performed in the higher-level language being compiled.
   - It is a non-trivial test of the language being compiled.
   - It is an inclusive consistency check as it must be capable of recreating its object code.

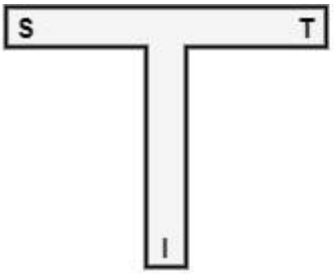
5. For bootstrapping, a compiler is defined by three languages −

   S→ Source language it compiles.

   T→ Target language it generates.

   I→ Implementation language that it is written in

   These languages can be represented using a T-diagram as



T - Diagram

which can be abbreviated as S_1T

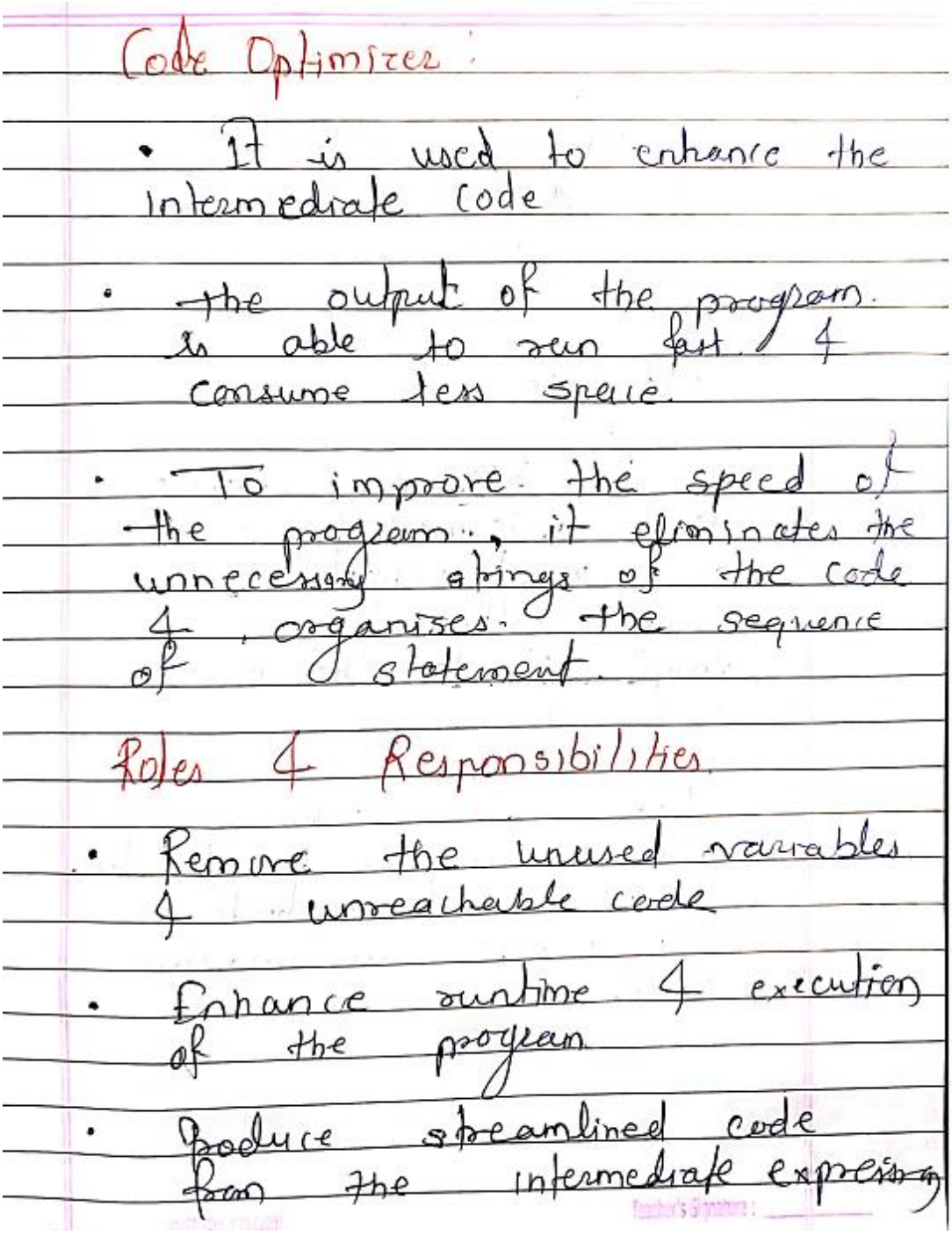
3. Just in time Compiler :

1. The Just-In-Time (JIT) compiler is an essential part of the JRE i.e. Java Runtime Environment, that is responsible for performance optimization of java based applications at run time.
2. The compiler is one of the key aspects in deciding the performance of an application for both parties i.e. the end-user and the application developer.
3. In order to improve performance, JIT compilers interact with the Java Virtual Machine (JVM) at run time and compile suitable bytecode sequences into native machine code.
4. While using a JIT compiler, the hardware is able to execute the native code, as compared to having the JVM interpret the same sequence of bytecode repeatedly and incurring overhead for the translation process.
5. The JIT compiler is able to perform certain simple optimizations while compiling a series of bytecode to native machine language.

2. b) Explain various phases of compilers comes under back end .

Code Optimizer – It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimization can be categorized into two types: machine-dependent and machine-independent.

Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.



Target Code Generator – The main purpose of the Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection, etc. The output is dependent on the type of assembler. This is the final stage of compilation. The optimized code is converted into relocatable machine code which then forms the input to the linker and loader.

Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.

Code Generator

It tries to acquire the intermediate code as input which is fully optimised. & map it to the machine code or lang.

Later the code generator helps in translating the intermediate code into the machine code.

Roles & Responsibilities

1) Translate the intermediate code to target machine code.

2) Select & allocate memory spots & register.

3. a) Explain Top Down Parser with example.

Parsing is classified into two categories, i.e. Top-Down Parsing, and Bottom-Up Parsing. Top-Down Parsing is based on Left Most Derivation whereas Bottom-Up Parsing is dependent on Reverse Right Most Derivation.

The process of constructing the parse tree which starts from the root and goes down to the leaf is Top-Down Parsing.

1. Top-Down Parsers constructs from the Grammar which is free from ambiguity and left recursion.
2. Top-Down Parsers uses leftmost derivation to construct a parse tree.
3. It does not allow Grammar With Common Prefixes.

Top-down parsing is a parsing technique in which the parser starts with the highest level of the grammar and works its way down to the lowest level. It is also called recursive descent parsing because the parser calls itself recursively to match the input against the grammar rules.

Here's an example of top-down parsing using the following grammar:

S → AaB A → c B → d

Suppose we have the following input: "cd".

The top-down parser will start with the start symbol S and try to match the input against the production rule S → AaB. It will then try to match A and B using the production rules A → c and B → d.

In this example, the top-down parser starts with the start symbol S and recursively applies the production rules to the input "cd". It successfully matches the input against the grammar rules and generates a parse tree.

The main advantage of top-down parsing is that it can be used to build predictive parsers that do not require backtracking. However, top-down parsing can be inefficient for large grammars, and it may require a significant amount of memory to store the parse stack.

Classification of Top-Down Parsing –

1. With Backtracking: Brute Force Technique

2. Without Backtracking:

1. Recursive Descent Parsing
2. Predictive Parsing or Non-Recursive Parsing or LL(1) Parsing or Table Driver Parsing

Recursive Descent Parsing –

1. Whenever a Non-terminal spend the first time then go with the first alternative and compare it with the given I/P String
2. If matching doesn't occur then go with the second alternative and compare with the given I/P String.
3. If matching is not found again then go with the alternative and so on.
4. Moreover, If matching occurs for at least one alternative, then the I/P string is parsed successfully.

LL(1) or Table Driver or Predictive Parser –

1. In LL1, first L stands for Left to Right and second L stands for Left-most Derivation. 1 stands for a number of Look Ahead tokens used by parser while parsing a sentence.
2. LL(1) parsing is constructed from the grammar which is free from left recursion, common prefix, and ambiguity.
3. LL(1) parser depends on 1 look ahead symbol to predict the production to expand the parse tree.
4. This parser is Non-Recursive.

3. b) Find the FIRST( ) and FOLLOW( )for the following grammar.
S→ aIJh, I → IbSe/c , J→ KLKr/∈ , K→ d/ ∈ , L→p/ ∈

Q. 3b)

→ FIRST(S) → FIRST(aIJh)
→ {a}

Here,
I → Ibse / c
eliminate the left recursion
∴ I → cI'
I' → bseI' / ∈

Therefore,
S → aIJh
I → cI'
I' → bseI' / ∈
J → KLKr / ∈
K → d / ∈
L → p / ∈

FIRST (I) → FIRST (cI') (small c)
→ {c}

FIRST (I') → FIRST (bseI') ∪ FIRST (∈)
→ {b, ∈}

FIRST (J) → FIRST (KLKr) ∪ FIRST (∈)
→ ----(1)

FIRST (K) → FIRST (d) ∪ FIRST (∈)
→ {d, ∈}

From (1) ⇒

FIRST (I) → FIRST (K) − {ϵ} ∪ FIRST (LKx) ∪ FIRST (ϵ) ... (2)

FIRST (L) → FIRST (P) ∪ FIRST (ϵ)
→ {p, ϵ}

From (2) ⇒

FIRST (I) → FIRST (K) − {ϵ} ∪ FIRST (LKx) ∪ FIRST (ϵ)
→ {d, ϵ} − {ϵ} ∪ FIRST (LKx) ∪ FIRST (ϵ) ... (3)

FIRST (LKx) → FIRST (L) − {ϵ} ∪ FIRST (Kx) ... (4)

FIRST (Kx) → FIRST (K) − {ϵ} ∪ FIRST (a)
→ {d, ϵ} − {ϵ} ∪ {x}
→ {d, x}

From (4) ⇒

FIRST (LKx) → {p, ϵ} − {ϵ} ∪ {d, x}
→ {p, d, x}

From (3) ⇒

FIRST (I) → {d, ϵ} − {ϵ} ∪ FIRST (LKx) ∪ FIRST (ϵ)
→ {d} ∪ {p, d, x} ∪ {ϵ}
→ {d, p, x, ϵ}

FOLLOW (I') = FOLLOW (I) ... Rule

- FOLLOW (I) → FIRST (Ih)
  → FIRST (I) - {ϵ} ∪ FIRST (h)
  → {d, p, r, ϵ} - {ϵ} ∪ {h}
  → {d, p, r, h}

as FOLLOW (I') = FOLLOW(I),
FOLLOW (I') → {d, p, r, h}

- FOLLOW(I) → FIRST (h)
  → {h}

- FOLLOW (k) → FIRST (LKr) ..... I → KLKr
  ⇒ {p, d, r}

- FOLLOW K → FIRST (r) ..... I → KLK r
  → {r}

- FOLLOW (L) → FIRST (Kr) ... I → KLKr
  → {d, r}

4. a) Show whether given grammar is LL(1) or not.

$S \rightarrow AaAb/BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

Ques. $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

→ FIRST (α) $\cap$ FIRST (β) $= \{\phi\}$ ---- ①

FIRST (AaAb) → FIRST (A) $- \{\epsilon\} \cup$ FIRST $\{aAb\}$

→ $\{\epsilon\} - \{\epsilon\} \cup \{a\}$

→ $\{a\}$

FIRST (BbBa) → FIRST (B) $- \{\epsilon\} \cup$ FIRST (bBa)

→ $\{\epsilon\} - \{\epsilon\} \cup \{b\}$

→ $\{b\}$

① ⇒ $\{a\} \cap \{b\} \rightarrow \{\phi\}$

FIRST (S) → FIRST (AaAb) $\cup$ FIRST (BbBa)

→ $\{a\} \cup \{b\}$

→ $\{a, b\}$

FIRST (A) → $\{\epsilon\}$

FIRST (B) → $\{\epsilon\}$

FIRST (S) → $\{\$\}$

$S \rightarrow AaAb$

FOLLOW (A) → FIRST (aAb) ; FIRST (b)

→ $\{a\}$ → $\{b\}$

FOLLOW (A) → $\{a, b\}$

$S \rightarrow BbBa$

FOLLOW (B) → FIRST (bBa) ; FIRST (a)

→ $\{b, a\}$

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$	

4. b) Explain why Top Down Parser is called left most derivative Parser.

Top-down parsing is sometimes called a left-most derivation parser because it produces a left-most derivation of the input string. A left-most derivation is a sequence of productions that generate the input string by replacing the left-most non-terminal symbol in the current string with its corresponding production rule until all non-terminal symbols are eliminated.

In a top-down parser, the parsing process starts with the start symbol of the grammar and applies the production rules in a left-to-right order, trying to match the input string. At each step, the parser chooses the left-most non-terminal symbol in the input string and applies the corresponding production rule to derive the next substring to be parsed.

For example, consider the following grammar:

S → aAB | bBA A → c | ε B → d

5. a) Find the reduced grammar equivalent to CFG

G = ({S,A,B,C } , { a,b,d} , S, P) where P contains

S → AC/SB

A→ bASC/a

B → aSB/bbC

C → Bc/ad

5. b) Consider the following Grammar

$E \rightarrow TA$

$A \rightarrow +TA/\epsilon$

$T \rightarrow FB$

$B \rightarrow *FB/\epsilon$

$F \rightarrow (E)/id$

Find FIRST( ) and FOLLOW( ) for each and every non terminal.

Q.5 b)

→ FIRST (E) → FIRST (TA) ---- ①

FIRST (A) → FIRST (+TA) ∪ FIRST (ϵ)
→ {+, ϵ}

FIRST (T) → FIRST (FB) ---- ②

FIRST (B) → FIRST (*FB) ∪ FIRST (ϵ)
→ {*, ϵ}

FIRST (F) → FIRST ((E)) ∪ FIRST (id)
→ {(, id}

Eqn ② ⇒ FIRST (T) → {(, id}

Eqn ① ⇒ FIRST (E) → {(, id}

FOLLOW (E) → FIRST ()) ∪ {$}
→ {), $}

FOLLOW (A) = FOLLOW (E)

FOLLOW (A) = {), $}

FOLLOW (T) = FIRST (A) − {ϵ} ∪ FOLLOW (A)
= {+, ϵ} − {ϵ} ∪ {), $}
= {+,), $}

FOLLOW (B) = {+,), $}

FOLLOW (F) = FIRST (B) − {ϵ} ∪ FOLLOW (B)
= {*, ϵ} − {ϵ} ∪ {+,), $}
= {*, +,), $}

6. a) Compare SLR , C LR and LALR Parser.

SLR Parser	LALR Parser	CLR Parser
It is very easy and cheap to implement.	It is also easy and cheap to implement.	It is expensive and difficult to implement.
SLR Parser is the smallest in size.	LALR and SLR have the same size. As they have less number of states.	CLR Parser is the largest. As the number of states is very large.
Error detection is not immediate in SLR.	Error detection is not immediate in LALR.	Error detection can be done immediately in CLR Parser.
SLR fails to produce a parsing table for a certain class of grammars.	It is intermediate in power between SLR and CLR i.e., SLR ≤ LALR ≤ CLR.	It is very powerful and works on a large class of grammar.
It requires less time and space complexity.	It requires more time and space complexity.	It also requires more time and space complexity.

S. no.	SLR Parsers	LALR Parsers	Canonical LR Parsers
1	SLR parser are easiest to implement.	LALR parsers are difficult to implement than SLR parser but less than CLR parsers.	CLR parsers are difficult to implement.
2.	SLR parsers make use of canonical collection of LR(0) items for constructing the parsing tables.	LALR parsers LR(1) collection , items with items having same core merged into a single itemset.	CLR parsers are uses LR(1) collection of items for constructing the parsing tables part.
3	SLR parsers don't do any lookahead i.e., they lookahead zero	LALR parsers lookahead one symbol.	CLR parsers lookahead one symbol.
4.	SLR parsers are cost effective to construct in terms of time and space.	The cost of constructing LALR parsers is i	CLR parsers are expensive to construct in terms – of time and space.

S. no.	SLR Parsers	LALR Parsers	Canonical LR Parsers
		intermediate between SLR and CLR parser.	
5.	SLR parsers have hundreds of states.	LALR parsers have hundreds of state and is same as number states in SLR parsers.	CLR parses have thousands of states.
6.	SLR parsers uses FOLLOW information to guide reductions.	LALR parsers uses lookahead symbol to guide reductions.	CLR parsers uses lookahead symbol to guide reductions.
7.	SLR parsers may fail to I produce a table for certain class of grammars on which ether succeed.	LALR parser works on very large class grammars.	CLR parser works on very large class of grammar.
8.	Every SLR(1) grammar is LR(1) grammar and LALR(1).	Every LALR(1) grammar may not be SLR(1) but every LALR(1) grammar is LR(1) grammar.	Every LR(1) grammar may not be SLR(1) grammar.
9.	A shift-reduce or reduce-reduce conflict may arise in SLR parsing table.	A shift-reduce conflict can not arise but a reduce-reduce conflict may arise.	A shift-reduce or reduce-reduce conflicted may arise but chances are less than that in SLR parsing tables.
10.	SLR parser is least powerful.	A LALR parser is intermediate in power between SLR and LR parser.	A CLR parsers is most powerful among the family canonical of bottom-up parsers.

6. b) Consider the following grammar:

$S \rightarrow aSbS / bSaS / \epsilon$

a) Show that this grammar is ambiguous by constructing two different leftmost derivation for the sentence abab.

7. a) consider the following grammar

S → ABC

A→ a/ϵ

B→ r/ϵ

C→ b/ϵ

Construct parsing table with LL(1) Parser

Q.7a)

→ FIRST (S) → FIRST (ABC) ----①

FIRST (A) → FIRST (a) ∪ FIRST (ϵ)

→ {a, ϵ}

FIRST (B) → FIRST (x) ∪ FIRST (ϵ)

→ {x, ϵ}

FIRST (C) → FIRST (b) ∪ FIRST (ϵ)

→ {b, ϵ}

from equ^n ① ⇒

FIRST (S) → FIRST (ABC)

∴ FIRST (S) → FIRST (A) − {ϵ} ∪ FIRST (BC) ----②

Now,

FIRST (BC) → FIRST (B) − {ϵ} ∪ FIRST (C)

→ {x, ϵ} − {ϵ} ∪ {b, ϵ}

∴ FIRST (BC) → {x, b, ϵ}

putting FIRST (BC) in equ^n ②

∴ FIRST (S) → FIRST (A) − {ϵ} ∪ FIRST (BC)

→ {a, ϵ} − {ϵ} ∪ {x, b, ϵ}

∴ FIRST (S) → {a, x, b, ϵ}

Now,

FOLLOW (S) → {$}

FOLLOW(A) → FIRST(BC) - {ε} ∪ FOLLOW(S)

→ {a, b, ε} - {ε} ∪ {$}

→ {a, b, $}

FOLLOW(B) → FIRST(C) - {ε} ∪ FOLLOW(S)

→ {b, ε} - {ε} ∪ {$}

→ {b, $}

FOLLOW(C) → FOLLOW(S)

→ {$}

	a	x	b	$
S	S → ABC	S → ABC	S → ABC	
A	A → a	A → ε	A → ε	A → ε
B		B → x	B → ε	B → ε
C			C → b	C → ε

7. b) Consider the following grammar

E → CC

C→ cC

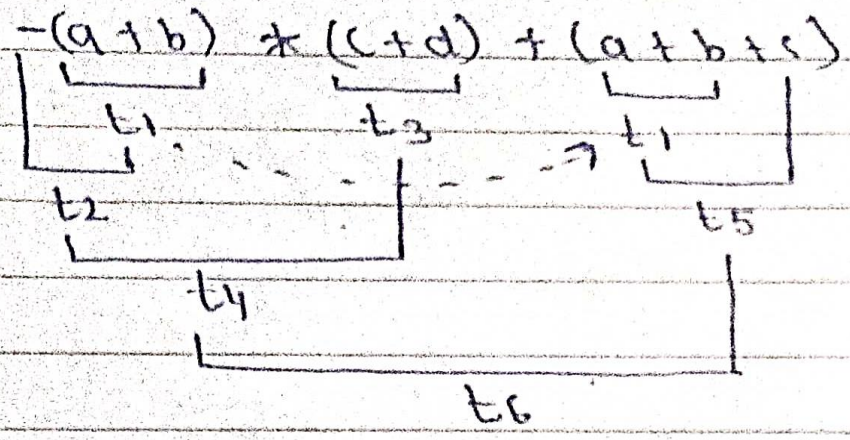
C→ d

Construct LR(0) Parser

8. a) Show Quadruple , Triple and Indirect triples for the following expression .

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

(Q.8a)



From above we get,

$t_1 = a + b$

$t_2 = -t_1$

$t_3 = c + d$

$t_4 = t_2 * t_3$

$t_5 = t_1 + c$

$t_6 = t_4 + t_5$

Quadruple Representation :→

	Operator	Operand 1	Operand 2	Result
1)	+	a	b	t_1
2)	-	t_1		t_2
3)	+	c	d	t_3
4)	*	t_2	t_3	t_4
5)	+	t_1	c	t_5
6)	+	t_4	t_5	t_6

Triple Representation :→

	Operator	operand 1	operand 2
1)	+	a	b
2)	-	(1)	
3)	+	c	d
4)	*	(2)	(3)
5)	+	(1)	c
6)	+	(4)	(5)

Indirect Triple Representation :→

pointer	List
(1)	(1)
2)	(2)
3)	(3)
4)	(4)
5)	(5)
6)	(6)

8. b) Find the TAC for following code :-

if (B > D and A < C) then P= a+1 else Q= b+1;

9. a) For the given grammar :

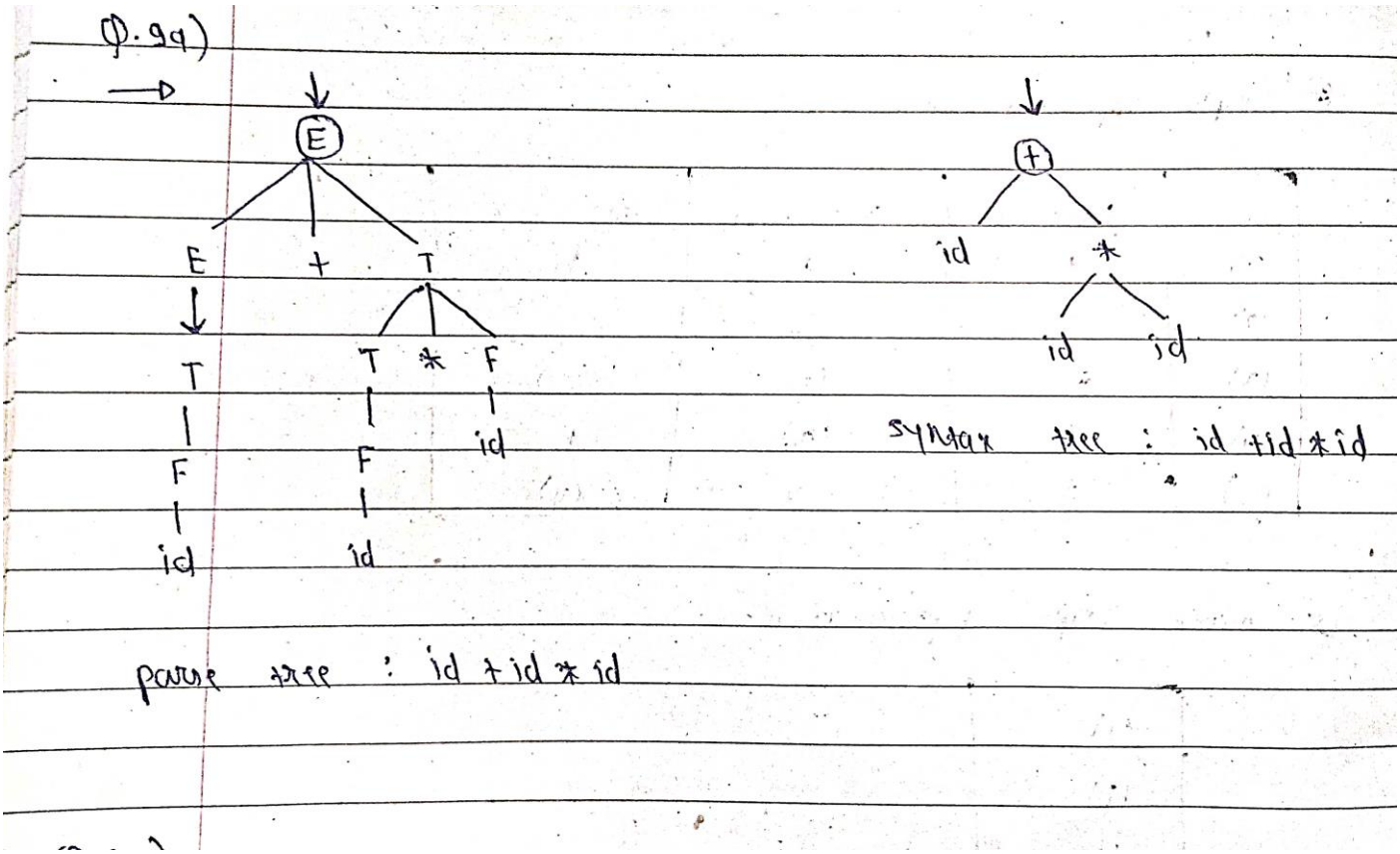
E-> E + T/T

T-> T * F/F

F-> (E)/id

Construct parse tree and syntax tree for string

W= id + id * id



9. b) Differentiate between Synthesized Attributes and Inherited Attributes.

S.NO	Synthesized Attributes	Inherited Attributes
1.	An attribute is said to be Synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.	An attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings node.
2.	The production must have non-terminal as its head.	The production must have non-terminal as a symbol in its body.
3.	A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself.	A Inherited attribute at node n is defined only in terms of attribute values of n's parent, n itself, and n's siblings.
4.	It can be evaluated during a single bottom-up traversal of parse tree.	It can be evaluated during a single top-down and sideways traversal of parse tree.
5.	Synthesized attributes can be contained by both the terminals or non-terminals.	Inherited attributes can't be contained by both, It is only contained by non-terminals.
6.	Synthesized attribute is used by both S-attributed SDT and L-attributed SDT.	Inherited attribute is used by only L-attributed SDT.
7.	EX:- E.val -> F.val  	EX:- E.val = F.val