Star

Date

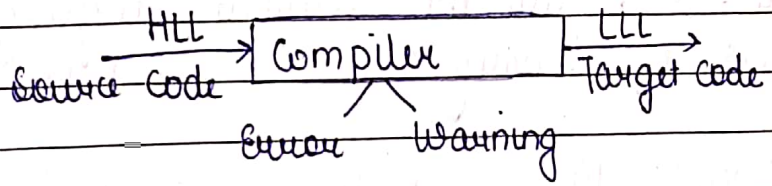
Page No.

Unit-1

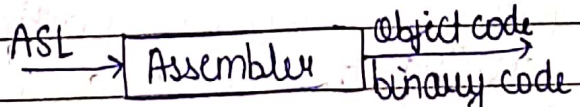
#) **Translator**-
A software system which convert the source code from one form of language to another form of language is called translator

Types of translator :-
There are two types of translator
1. Compiler
2. Assembler

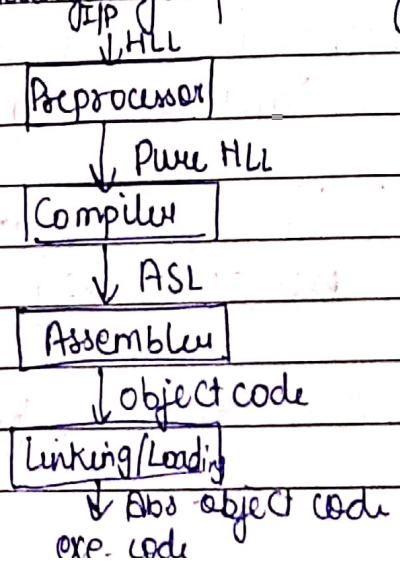
#) **Compiler**- A translator which convert the source code from high level language to low level language is called compiler.



#) **Assembler**- A assembler is a software which converts assembly code into object code or binary code is called assembler.



Functions of language processing system (LPS):-



Date Star
Page No.

Object code

Relocatable Absolute object code

i) Preprocessor :- It includes all header files and also evaluates if any MACRO is included. The preprocessor is also called as MACRO Evaluator. Preprocessing is optional i.e. if any language which does not support # include and MACRO's preprocessor is not required.

ii) Compiler :- It takes the preprocessor as input and converts into assembly code (low level lang.)

iii) Assembler :- It convert the assembly language into object code or binary code or machine code

iv) linking and loading provides four functions
a) Allocation c) linker
b) Relocation d) loader

Allocation - Getting the memory portions from operating system and storing the object data or object code.

Relocation - Mapping the relative address to the physical address and relocating the object code.

linker - Combine all the executable object module to one single executable file.

loader - loading the executable file into permanent

Star
Date
Page No.

#) Difference Between Compiler & Interpreter

Compiler	Interpreter
1) A compiler translates the entire source code in one run	1) An interpreter translates the entire source code one line by line.
2) It consumes less time.	2) It consumes more time.
3) It is more efficient	3) It is less efficient
4) CPU utilization is more	4) CPU utilization is less as compared to compiler
5) The Compiler is larger	5) The Interpreter are smaller than compiler.
6) It is not flexible	6) It is flexible
7) The localization of errors is difficult	7) The localization of error is easier then the compiler
8) Both Syntatic & Semantic errors can be checked simultaneously	8) Only syntatic errors are checked.

#) Design of Compiler :-

I/P
↓
Lexical Analysis
↓
Syntax Analysis
↓
Symantic Analysis
↓
Intermediate Code generation
↓
Code optimization
↓
Target Code
↓
O/P

Date
Page No.

Phases of compiler :-

Phases - Converting the source code from one form of representation to another form of representation is called phase.

The phases can be divided into two parts :-

i) Analysis phase
ii) Synthesis phase

Analysis phase - It breaks the source program into constituent pieces and imposes a grammatical structure on them which further uses the structure to create an intermediate representation of the source program.

- It is also termed as front end of compiler.
- Information about the source program is collected and stored in a data structure called symbol table

Synthesis Phase -

- Synthesis part takes the intermediate representation as input and transforms it to the target program
- It is also termed as back end of compiler

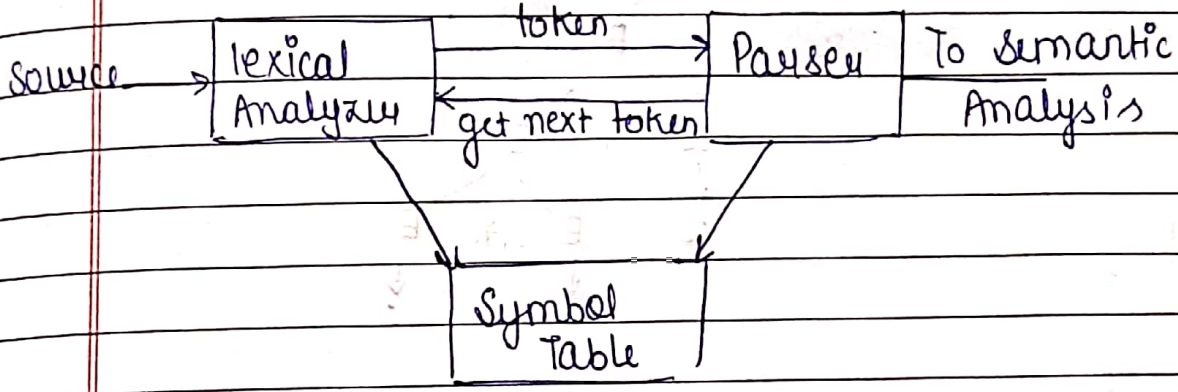
The design of compiler can be decomposed into :-

1) Lexical Analysis
2) Syntax Analysis
3) Semantic Analysis
4) Intermidiate Code generation
5) Code optimization
6) Code Generation

Star
Date
Page No.

1) Lexical Analysis :-

- Lexical Analysis is the first phase of compiler which is also termed as scanning.
- Reads the input characters and those characters are grouped to form a sequence called lexems which produces token as output uses for syntax analysis
- Stripes out source program comments & white space in form of blank, tab.



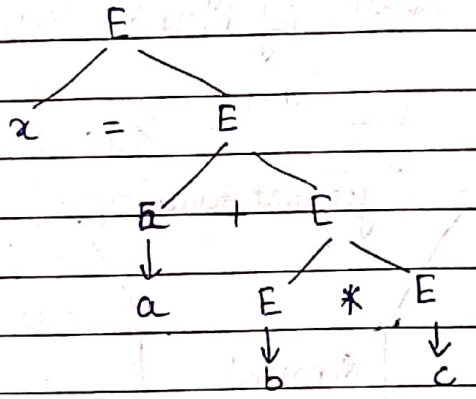
- Token: Token is a sequence of characters that represent lexical unit which matches with the pattern, such as keywords, operators, identifiers etc

- Lexeme: It is a sequence of characters in the source program that matches the pattern for a token

- Pattern: Pattern describes the rule that the lexemes of a token takes. It is structure that must be matched by strings.
- Once a token is generated the corresponding entry is made in the symbol table

Date
Page No.

2) **Syntax Analysis** :-
Syntax Analyzer verify the grammatical mistake of the source code. To verify the syntax of the code the language must be defined by CFG. Syntax Analyzer take the stream of tokens as I/P & generates the parse tree
eg → $x = a + b * c$
Parse Tree :-



3) **Semantic Analysis** :-
Symantic Analysis verify the meaning of each and every sentance by performing type checking Syntax Analyzer just verifies whether the operator is operating on required no. of operands or not and does not looking into working

4) **Intermediate Code Generator** :-
The source code is converted into intermediate representation to make code generation process simple & easy and to achieve the platform independently.

5)

6)

Que Ans

Star
Date
Page No.

Ex→

$x = a + b * c$

$t_1 = b * c$
$t_2 = a + t_1$
$x = t_2$
} Intermediate code

5) Code optimization

Reducing the number of instructions without affecting the outcome of the source program is called optimization

It is of two types :-

(i) Machine independent optimization
(ii) Machine dependent optimization

6) Target Code

The optimizationed source code should converted into assembly code

Ex

$x = a + b * c$

```
MUL  R1  R2  // b*c        R0 → a
ADD  R0  R2  // a+b*c       R1 → b
MOVE R2  x   // x=a+b*c     R2 → c
```

Que Why is Intermediate Code Generator is required?

Ans If we generate machine code directly from source code, we may face 2 problems

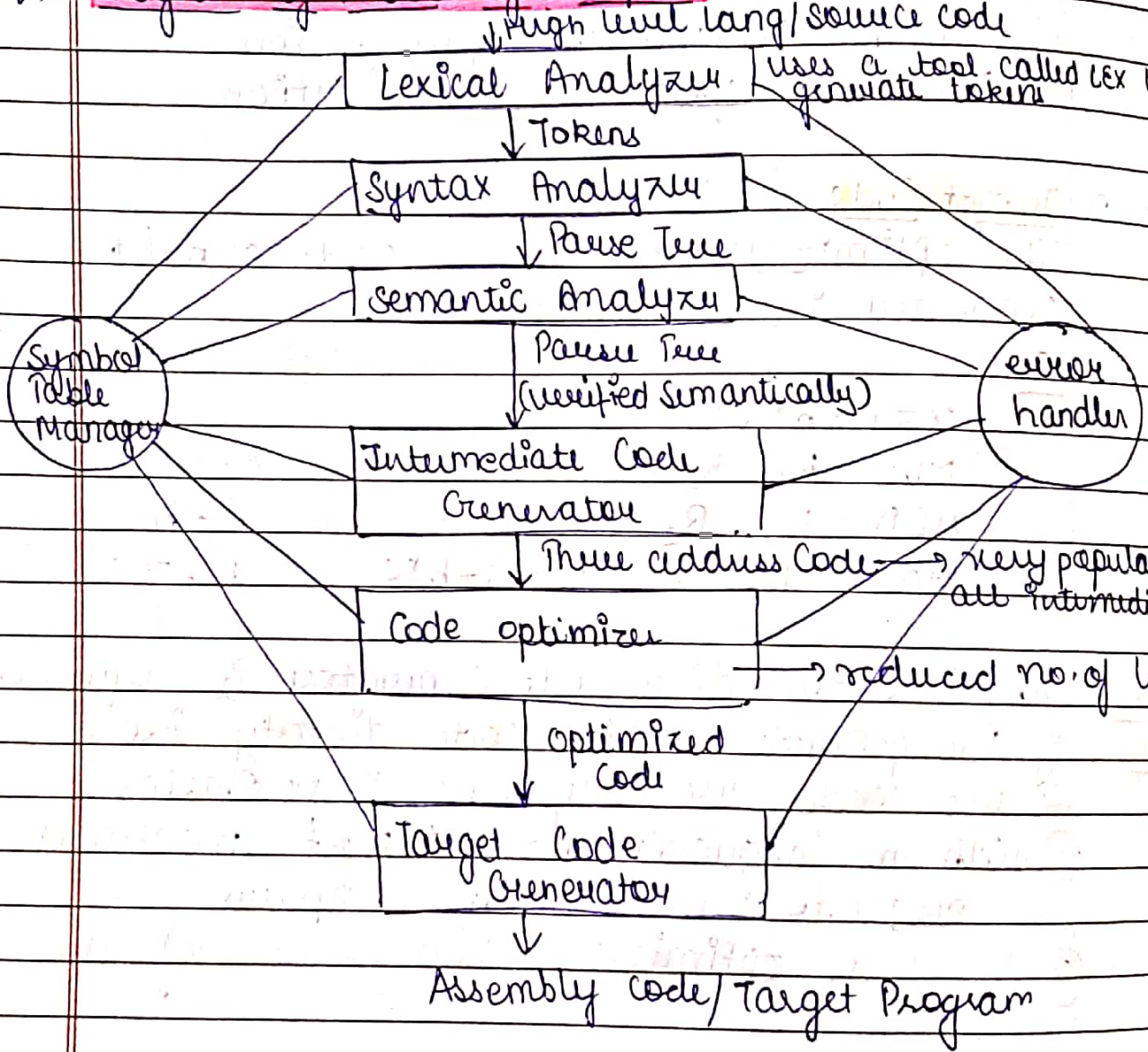
(1) with m languages and n target machines, we need to write m×n compilers

(2) The Code optimizer which is one of the

Star
Date
Page No.

longest and very difficult to write components of the compiler can't be reused.
By converting source code to an intermediate code, a machine independent optimisation can be done.
An intermediate code must be easy to produce and easy to translate to machine code, shouldn't contain any machine specific codes is a sort of universal assembly language.

#) diagram of Phases of Compiler



Star
Date
Page No.

#) **Lexical Analysis**

A lexical Analyzer takes a high level programming language as input in the form of sequence of characters and produce a sequence of tokens by removing any white space chars and comment line.

So, the LA produce error for illegal symbols. In such cases, it skips character in the input until a wellformed token is found.

Q. What is Token?

A. Token is a sequence of characters that represent lexical unit which matches with the pattern. Ex- float, identifier, =, +, ;

Q. What is pattern?

A. The set of strings for which some token is produced ex- float, =, +, ;

Q. What is Lexeme?

A. The sequence of characters match by pattern to form the corresponding token.
ex- "float", "silicon", "=", "+", ";"

* Each token consisting of a token name & an optional attribute value.

* Regular definition (modified version of regular expressions) are used for specification of

Star
Date
Page No.

tokens & from these definitions, LA can be generated automatically using LEX.

Functions of lexical Analyzer

1) Read input characters from source program
2) Dividing the programs into valid tokens (tokanization)
3) Remove white spaces characters
4) Remove Comments (//, #)
5) Helps to Identify tokens into the symbol table
6) It Generates lexical Errors

Lexical Error :-

A character sequence which is not possible to scan any valid token is called a lexical error
lexical phase error can be :-

- → Spelling error
- → Exceeding length of identifier
- → Appearance of illegal character

eg- x = 1x ab
↓
neither a no. nor a identifier

* lexical phase error is found during the execution of the program

Advantages

- Lexical Analyzer method is used by programs like compiler which can use the parsed data from
- It is used by web browser to format and display a web page with the help of parsed

data from javascript, HTML, CSS.
- It helps to construct a specialized and potentially more efficient processor for a task

Disadvantages of lexical Analyzer

- You need to spend significant time reading the source program and partitioning it in the form of tokens
- Some Regular Expression are quite difficult to understand compared to PEG or EBNF rules.
- More Effort is needed to develop & debug the lexer and its token descriptions

#) Specifications of Tokens

There are the types of specifications of tokens :-

1) Alphabets - Any finite set of symbols {0,1} is a set of binary alphabets {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of hexadecimal alphabets {a-z, A-Z} is a set of English language alphabets

2) Strings - Any finite sequence of alphabets is called a strings. length of the string is the total no. of occurence of alphabets e.g. the length of the string is the total number of occurence of alphabet.

3) Language - A language is considered as a finite set of alphabets. Computer languages are considered as finite sets and mathematically set operations can be performed on them. Finite language can be described by means of regular Expressions.

operations on languages :-

- Union of two languages L & M is written as
  $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
- Concatenation of two languages L & M is written as $LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
- The Kleene Closure of a language L is written as
  L^* = zero or more occurrence of language L

4) Regular Expression - Regular expression have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expression is known as regular grammar. The languages defined by regular Grammar is known as Regular language.
Regular languages are easy to understand and have efficient implementation.

Star
Date
Page No.

Lexical Analysis Generator (LEX)

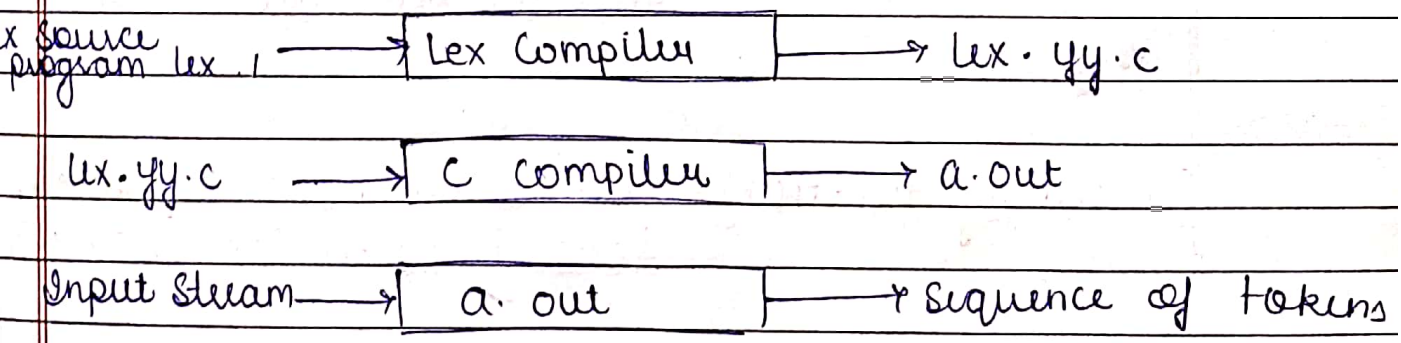
→ LEX is a tool / computer program which generates lexical Analyzer. It is used by YACC parser / Generator.

→ LEX written by Mike, Lesk and Eric Schmidt and describe in 1975

→ Its main purpose is to break up an input stream into tokens. (sequence of tokens)

→ It reads the input stream and produces the source code as output through implementing the lexical analyzer.

→ It is a tool for automatically generating a lexer.



→ Lex can be used with a parser Generator to perform lexical analysis.

Function of Lex through diagram :-

1) Firstly lexical analyzer creats a program lex.l in the lex language. Then lex compiler runs the lex.l program and produces a C program lex.yy.c

2) Finally C compiler runs the lex.yy.c program and produces an object program a.out
3) a.out is a lexical analyzer that transforms an input stream into a sequence of tokens.

Structure of LEX :-
A lex program is seperated into three sections by %. % delimeters.

{declaration} ⇒ declaration of variable, constant
%. %.
{translation rules} ⇒ have the pattern {action} contains rules in the form of regular expressions
%. %

{auxillary function} ⇒ can be compiled seperately & loaded with lex program

4) Symbol Table Management

Symbol Table is an important data structure created and maintained by compilers in order to store information about the occurence of various entities such as variable names, function names, objects, classes etc.

Star
Date
Page No.

Symbol table is used by both analysis & synthesis parts of a compiler. It is used for scope management.

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

#) Finite Automata

- Finite Automata are used to recognize patterns
- It takes string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.
- At the time of transition, the automata either move to the next state or stay in the same state.
- Finite Automata has two state, Accept State or Reject State
- Finite State Automata are used to design L.A.
- F.S. Automata are machine that accepts regular languages.

Date
Page No.

A Finite Automata is a collection of five-tuple $(Q, \Sigma, \delta, q_0, F)$ where :-
Q is finite set of States
Σ is finite set of the input Symbol
q_0 is initial State
F is final State
δ is transition function
It is of two types :-
1) DFA 2) NFA

1) DFA (Deterministic Finite Automata)

- DFA refers to when the machine is reading an input string one symbol at a time
- In DFA, there is only one path for specific input from the current state to next state
- DFA does not accept the null move
- DFA can contain multiple final states. It is used in lexical Analysis in Compiler
- It consist of 5 tuples $(Q, \Sigma, q_0, F, \delta)$.
- Transition function can be defined as :-

$$\delta : Q \times \Sigma \rightarrow Q$$

2) NFA (Non-Deterministic Finite Automata)

- It is easy to construct an NFA than DFA for a given Regular Language.
- The finite automata are called NFA when there exist many paths for specific input

from current state to next state
- Every NFA is not DFA but each NFA can be translated into DFA.
- It accepts Null move
- It contains multiple next states.

Transition function → $\delta : Q \times \Sigma \rightarrow 2^Q$

It contains five tuples $(Q, \Sigma, q_0, F, \delta)$

#) Three Address Code

Three Address Code is a simple, intermediate code to construct and convert to machine code. It employs just three places and one operator to describe an expression and the value computed at each instruction is saved in a Compiler-Generated temporary variable

- It is used for by the optimizing compilers.
- The given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
- It is a combination of assignment and a binary operator.
- The three address code has three operands.

General Representation :-

$a = b \; op \; c$

operator

a, b, or c are operands

① **Quadruple** -

It is structure with consist of 4 fields namely op, arg1, arg2, and result

op denotes operator

arg1, arg2 denotes operands

result is to store the result of expression

operator
arg1
arg2
result

Advantages

1) Easy to rearrange code for global optimization
2) One can quickly access value of temporary variables using symbol table

Disadvantages

1) Contain lot of temporaries
2) Temporary variable creation increases time and space complexity

Ex →

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

The three address code :-

$t_1 = \text{uminus } c$

$t_2 = b^* t_1$

$t_3 = \text{uminus } c$

$t_4 = b^* t_3$

$t_g = t_2 + t_4$
$a = t_5$

#	op	Arg 1	Arg 2	Result
0	uminus	c		t_1
1	*	t_1	b	t_2
2	uminus	c		t_3
3	*	t_3	b	t_5
4	+	t_2	t_4	t_5
5	=	t_5		a

Quadruple Representation

② **Triples** -
This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of three fields namely op, arg1, arg2

Disadvantages -

1) Temporaries are implicit and difficult to rearrange code
2) It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also.

ex→

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

#	op	Arg 1	Arg 2
0	uminus	c	
1	*	0	b
2	uminus	c	
3	*	2	b
4	+	1	3
5	=	a	4

#) **Difference Between Static and Dynamic Memory Allocation**

Static Allocation	Dynamic Allocation
1) When the allocation of memory performs at the compile time, then it is known as static memory	1) When the memory allocation is done at the execution or run time then it is called dynamic memory allocation
2) The memory is allocated at the compile time	2) The memory is allocated at the runtime
3) While execution, the memory cannot be changed	3) While execution, the memory can be changed
4) It saves running time as it is fast	4) It is slower than static memory allocation
5) It allots memory from the stack	5) It allots memory from the heap

6) It is less Efficient	6) It is more Efficient
7) The memory allocation is simple	7) The memory allocation is complicated.
8) It is preferred in array	8) It is preferred in linked list.
9) Once the memory is allocated, it will remain from the beginning to end of program	9) The memory can be allocated at any time in the program.

4) Optimizing Codes

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 so that faster running machine code will result.

Code optimization 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.

Date Star
Page No.

#) Machine Independent Optimization :-

- Machine Independent optimization attempts to improve the intermediate code to get a better target code. The part of the code which is transformed here does not involve any absolute memory location or any CPU register.
- The process of intermediate code generation has much inefficiency like : using variable instead of constants, extra copies of variable, repeated evaluation of expression.
- Through code optimization, you can remove such efficiencies and improves code
- It can change the structure of program sometimes of beyond recognition like : unrolls loops, inline functions, eliminates some variables that are programmer defined

#) Machine Dependent Optimization :

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

Star
Date
Page No.

#) Peephole optimization

This optimization technique works locally on the source code to transform it into an optimized code. By locally, it means a small portion of the code block at hand. These methods can be applied on intermediate code as well as target code. A bunch of statement is analyzed and are checked for following optimization :-

1) Ridundant Instruction elimination -
At the compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even some of are removed

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

3) Flow of Control Optimization
There are instances in code where the program control jumps back and forth without performing any significant task. These jumps can be removed.

Date
Page No.

4) Algebric Expression Simplication
There are occasions where algebric expression can be made simple.

5) Strength Reduction
There are operations that consume more space And time. Their strength can be reduced by replacing them with other operations that consume less time and space, but produce same result.

6) Accessing Machine Instructions
The target machine can deploy more sophisticated and instructions which can have the capability to perform specific operations much efficiently. If the target code can accomdate these instruction directly, that will only improve the quality of code but also yield more efficient result.