PVG's
COLLEGE OF ENGINEERING
Nashik

Department of Computer Engineering

LABORATORY MANUAL

2018-2019

LABORATORY PRACTICE-IV

[Compiler]

BE-COMPUTER ENGINEERING

SEMESTER-II

Subject Code: **410255**

TEACHING SCHEME

Practical: 4 Hrs/Week

EXAMINATION SCHEME

Oral: 50 Marks

Term Work: 50 Marks

-: Name of Faculty:-

Prof. A.R.Jain

Asst. Professor, Department of Computer Engineering.

Assignment No.	1
Title	Implement a Lexical Analyzer using LEX for a subset of C. Cross check your output with Stanford LEX.
Roll No.	
Class	B.E. (C.E.)
Date	
Subject	Laboratory Practice-IV
Signature	

Assignment No: 1

Title: Implement a Lexical Analyzer using LEX for a subset of C. Cross check your output with Stanford LEX.

Aim:

Assignment to understand the syntax of LEX specifications, built-in functions and variables.

Objectives:

- To understand first phase of compiler: Lexical Analysis.
- To learn and use compiler writing tools.
- Understand the importance and usage of LEX automated tool.

Theory:

Introduction:

LEX stands for Lexical Analyzer.LEX is a UNIX utility which generates the lexical analyzer. LEX is a tool for generating scanners. Scanners are programs that recognize lexical patterns in text. These lexical patterns (or regular expressions) are defined in a particular syntax. A matched regular expression may have an associated action. This action may also include returning a token. When Lex receives input in the form of a file or text, it attempts to match the text with the regular expression. It takes input one character at a time and continues until a pattern is matched. If a pattern can be matched, then Lex performs the associated action (which may include returning a token). If, on the other hand, no regular expression can be matched, further processing stops and Lex displays an error message. Lex and C are tightly coupled. A lex file (files in Lex have the .l extension eg: first.l) is passed through the lex utility, and produces output files in C (lex.yy.c). The program lex.yy.c basically consists of a transition diagram constructed from the regular expressions of first.l These file is then compiled object program a.out, and lexical analyzer transforms an input streams into a sequence of tokens as show in fig 1.1.

To generate a lexical analyzer two important things are needed. Firstly it will need a precise specification of the tokens of the language. Secondly it will need a specification of the action to be performed on identifying each token.

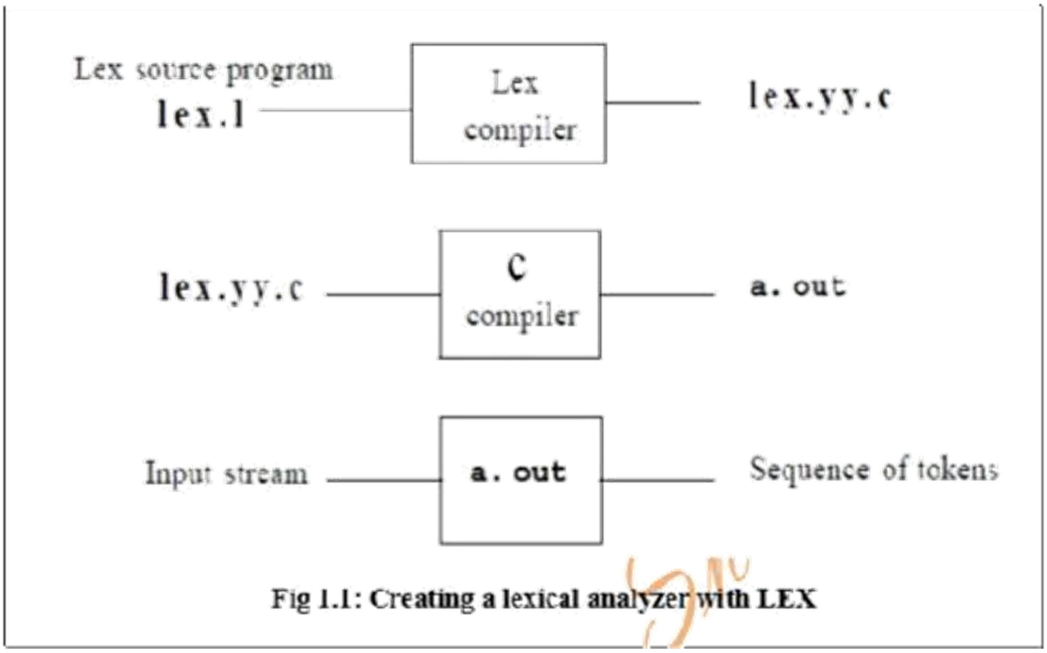


Fig 1.1: Creating a lexical analyzer with LEX

1. LEX Specifications:

The Structure of lex programs consists of three parts:

```
%{
        Definitions Section
%}
%%
        Rules Section
%%
        Auxiliary Function
```

Definition Section :

The Definition Section includes declarations of variables, start conditions regular definitions, and manifest constants (A manifest constant is an identifier that is declared to represent a constant e.g. # define PIE 3.14).

- C code: Any indented code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.
- Definitions: A definition is very much like # define cpp directive. For example

letter [a-zA-Z]+

digit [0-9]+

These definitions can be used in the rules section: one could start a rule

```
{letter}{printf("n Wordis = %s",yytext);}
```

- State definitions: If a rule depends on context, it‟s possible to introduce states and incorporate those in the rules. A state definition looks like %s STATE, and by default a state INITIAL is already given.

Rule Section:

Second section is for translation rules which consist of regular expression and action with respect to it. The translation rules of a Lex program are statements of the form:

```
p1 {action 1}
p2 {action 2}
p3 {action 3}
...    ...
...    ...
pn {action n}
```

Where, each p is a regular expression and each action is a program fragment describing what action the lexical analyzer should take when a pattern p matches a lexeme. In Lex the actions are written in C.

Auxiliary Function(User Subroutines):

Third section holds whatever auxiliary procedures are needed by the actions. If the lex program is to be used on its own, this section will contain a main program. If you leave this section empty you will get the default main as follow:

```
int main()
{
    yylex();
    return 0;
}
```

In this section we can write a user subroutines its option to user e.g. yylex() is a unction automatically get called by compiler at compilation and execution of lex program or we can call that function from the subroutine section.

2. Built - in Functions:

No.	Function	Meaning
1	yylex()	The function that starts the analysis. It is automatically generated by Lex.
2	yywrap()	This function is called when end of file (or input) is encountered. If yywrap() returns 0, the scanner continues scanning, while if it returns 1 the scanner returns a zero token to report the end of file.
3	yyless(int n)	This function can be used to push back all but first „n“ characters of the read Token.
4	yymore()	This function tells the lexer to append the next token to the current token.
5	yyerror()	This function is used for displaying any error message.

3. Built - in Variables:

No.	Variables	Meaning
1	yyin	Of the type FILE*. This point to the current file being parsed by the lexer. It is standard input file that stores input source program.
2	yyout	Of the type FILE*. This point to the location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output.
3	yytext	The text of the matched pattern is stored in this variable (char*) i.e. When lexer matches or recognizes the token from input token the lexeme stored in null terminated string called yytext. OR This is global variable which stores current token
4	yyleng	Gives the length of the matched pattern. (yyleng stores the length or number of character in the input string)The value in yyleng is same as strlen() functions.
5	yylineno	Provides current line number information. (May or may not be supported by the lexer.)
6	yylval	This is a global variable used to store the value of any token.

1. Regular Expression:

No.	RE	Meaning
1	a	Matches a
2	abc	Matches abc
3	[abc]	Matches a or b or c
4	[a-f]	Matches a,b,c,d,e or f
5	[0-9]	Matches any digit
6	X^+	Matches one or more of x
7	X*	Matches zero or more of x
8	[0-9]+	Matches any integer
9	(…)	Grouping an expression into a single unit

10	|	Alteration (or)
11	(b|c)	Is euivalent to [a-c]*
12	X?	X is optional (0 or 1 occurrence)
13	If(def)?	Matches if or ifdef
14	[A-Za-z]	Matches any alphabetical character
15	.	Matches any character except new line
16	\.	Matches the . character
17	\n	Matches the new character
18	\t	Matches the tab character
19	\\	Matches the \ character
20	[\t]	Matches either a space or tab character
21	[^a-d]	Matches any character other than a,b,c and d
22	$	End of the line

2. **Steps to Execute the program:**

$ lex filename.l (eg: first.l)

$cc lex.yy.c–ll or gcc lex.yy.c–ll

$./a .out

Algorithm:

1.Start the program.

2.Lex program consists of three parts.

a. Declaration%%

b.Translation rules %%

c. Auxilary procedure.

3.The declaration section includes declaration of variables, maintest, constants and regular definitions.

4. Translation rule of lex program are statements of the form

a. P1 {action}

b. P2 {action}

c. …

d. …

e. Pn {action}

5. Write a program in the vi editor and save it with .l extension.

6.Compile the lex program with lex compiler to produce output file as lex.yy.c. eg $ lex filename.l $ cc lex.yy.c -ll

7.Compile that file with C compiler and verify the output.

Conclusion:

LEX is a tool which accepts regular expressions as an input & generates a C code to recognize that token. If that token is identified, then the LEX allows us to write user defined routines that are to be executed. When we give input specification file to LEX, LEX generates lex.yy.c file as an output which contains function yylex() which is generated by the LEX tool & contains a C code to recognize the token & action to be carried out if we find the token.

Assignment No.	2
Title	Implement a parser for an expression grammar using YACC and LEX for the subset of C. Cross check your output with Stanford LEX and YACC.
Roll No.	
Class	B.E. (C.E.)
Date	
Subject	Laboratory Practice-IV
Signature	

Assignment No: 2

Title: Implement a parser for an expression grammar using YACC and LEX for the subset of C. Cross b check your output with Stanford LEX and YACC.

Aim: Assignment to understand basic syntax of YACC specifications built-in functions and variables

Objective:

- To understand Second phase of compiler: Syntax Analysis.
- To learn and use compiler writing tools.
- Understand the importance and usage of YACC automated tool

Theory:

Parser generator facilitates the construction of the front end of a compiler. YACC is LALR parser generator. It is used to implement hundreds of compilers. YACC is command (utility) of the UNIX system. YACC stands for "Yet Another Compiler Complier".

File in which parser generated is with .y extension. e.g. parser.y, which is containing YACC specification of the translator. After complete specification UNIX command. YACC transforms parser.y into a C program called y.tab.c using LR parser. The program y.tab.c is automatically generated. We can use command with –d option as

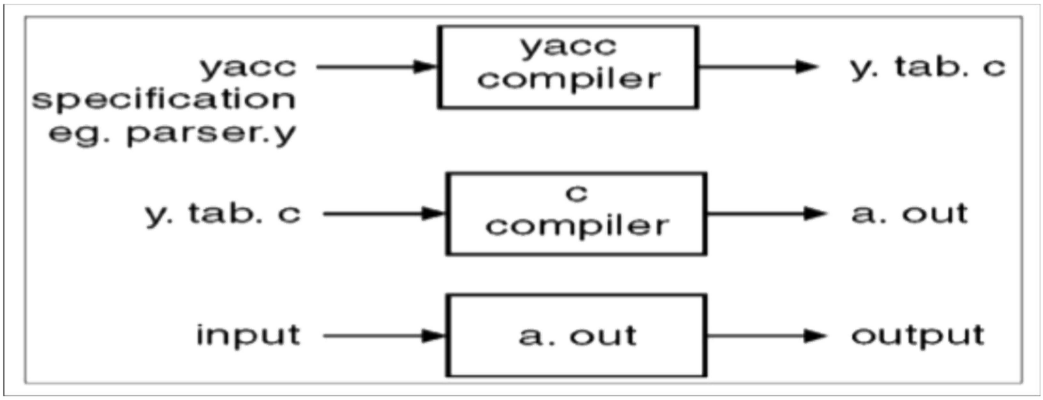
yacc –d parser.y

By using –d option two files will get generated namely y.tab.c and y.tab.h. The header file y.tab.h will store all the token information and so you need not have to create y.tab.h explicitly.

The program y.tab.c is a representation of an LALR parser written in C, along with other C routines that the user may have prepared. By compiling y.tab.c with the ly library that contains the LR parsing program using the command.

cc y tab c – ly

we obtain the desired object program a out that perform the translation specified by the original program. If procedure is needed, they can be compiled or loaded with y.tab.c, just as with any C program.



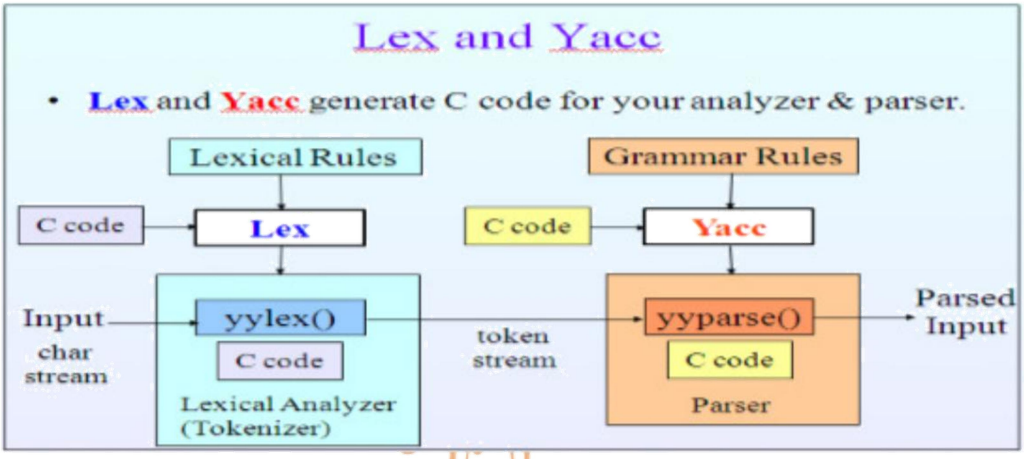


Fig: 3.1 YACC: Parser Generator Model

LEX recognizes regular expressions, whereas YACC recognizes entire grammar. LEX divides the input stream into tokens, while YACC uses these tokens and groups them together logically. LEX and YACC work together to analyze the program syntactically. The YACC can report conflicts or ambiguities (if at all) in the form of error messages.

1. YACC Specifications:

   The Structure of YACC programs consists of three parts:

```
%{
      Definitions Section
 %}
 %%
     Rules Section (Context Free Grammar)
 %%
       Auxiliary Function
```

Definition Section:

The definitions and programs section are optional. Definition section handles control information for the YACC-generated parser and generally set up the execution environment in which the parser will operate.

Declaration part:

In declaration section, %{ and %} symbol used for C declaration. This section is used for definition of token, union, type, start, associativity and precedence of operator. Token declared in this section can then be used in second and third parts of Yacc specification.

Translation Rule Section:

In the part of the Yacc specification after the first %% pair, we put the translation rules. Each rule consists of a grammar production and the associated semantic action. A set of productions that we have been writing:

<left side> → <alt 1> | <alt 2> | … <alt n>

Would be written in YACC as

```
<left side>  : <alt 1> {action 1}
             | <alt 2> {action 2}
             … … … … …
             | <alt n> {action n}
             ;
```

In a Yacc production, unquoted strings of letters and digits not declared to be tokens are taken to be nonterminals. A quoted single character, e.g. 'c', is taken to be the terminal symbol c, as well as the integer code for the token represented by that character (i.e., Lex would return the character code for ' c' to the parser, as an integer). Alternative bodies can be separated by a vertical bar, and a semicolon follows each head with its alternatives and their semantic actions. The first head is taken to be the start symbol.

A Yacc semantic action is a sequence of C statements. In a semantic action, the symbol $$ refers to the attribute value associated with the nonterminal of the head, while $i refers to the value associated with the ith grammar symbol (terminal or nonterminal) of the body. The semantic

action is performed whenever we reduce by the associated production, so normally the semantic action computes a value for $$ in terms of the $i's. In the Yacc specification, we have written the two E-productions.

E E + T/T

and their associated semantic action as:

```
exp :   exp „+“ term        {$$ = $1 + $3;}
    |   term
    ;
```

In above production exp is $1, „+“ is $2 and term is $3. The semantic action associated with first production adds values of exp and term and result of addition copying in $$ (exp) left hand side. For above second number production, we have omitted the semantic action since it is just copying the value. In general {$$ = $1;} is the default semantic action.

Supporting C-Routines Section:

The third part of a Yacc specification consists of supporting C-routines. YACC generates a single function called yyparse(). This function requires no parameters and returns either a 0 on success, or 1 on failure. If syntax error over its return 1.The special function yyerror() is called when YACC encounters an invalid syntax. The yyerror() is passed a single string (char) argument. This function just prints user defined message like:

```
yyerror (char err)
{
    printf (“Divide by zero”);
}
```

When LEX and YACC work together lexical analyzer using yylex () produce pairs consisting of a token and its associated attribute value. If a token such as DIGIT is returned, the token value associated with a token is communicated to the parser through a YACC defined variable yylval. We have to return tokens from LEX to YACC, where its declaration is in YACC. To link this LEX program include a y.tab.h file, which is generated after YACC compiler the program using – d option.

2. Built-in Functions:

Function	Meaning
yyparser()	This is a standard parse routine used for calling syntax analyzer for given translation rules. When yyparse() is call, the parser attempts to parse an input stream.
yyerror()	This function is used for displaying any error message when a yacc detects a syntax error

3. Built-in Types:

Type	Meaning
%token	Used to declare the tokens used in the grammar. Eg.:- %token NUMBER
%start	Used to declare the start symbol of the grammar. Eg.:- %start S Where S is start symbol
%left	Used to assign the associatively to operators. Eg.: %left „+“ „-„ -Assign left associatively to + & – with lowest precedence. %left „*“ „/„ -Assign left associatively to * & / with highest precedence.
%right	Used to assign the associatively to operators. Eg.: %right „+“ „-„ -Assign right associatively to + & – with lowest precedence %right „*“ „/„ -Assign right associatively to * & / with highest precedence.
%nonassoc	Used to unary associate. Eg.:- %nonassoc UMINUS
%prec	Used to tell parser use the precedence of given code. Eg.:- %prec UMINUS
%type	Used to create the type of a variable. Eg.:- %type <name of any variable> exp
%union	Token data types are declared in YACC using the YACC declaration % union, like this : % union { char str ; int num ; }

4. Special Characters:

Characters	Meanings
%	A line with two percent signs separates the part of yacc grammar. All declarations in definition section start with %, including %{ %},%start, %token, %type, %left, %right, %nonassoc and %union.
$	In action, a dollar sign introduces a value references e.g: $3 value of the third symbol in the rule‟s right-hand side.
‘ ’	Literal tokens are enclosed in single quotes. Eg: „+‟ or „-„ or „*‟ or „\‟ etc.
< >	In value references in an action, you can override the value‟s defaults type by enclosing the type name in angle brackets.
{ }	The C code in action is enclosed in curly braces
;	Each rule in rules section should end with semicolon, except those that are immediately followed by rule that starts a vertical bar.
|	When two consecutive rules have same left-hand side, the second rule is separated by vertical bar.
:	In rule section, colon is used to separate left-hand side and right-hand side.

5. Steps to Execute the program

$ lex filename.l (eg: cal.l)

$ yacc -d filename.y (eg: cal.y)

$cc lex.yy.c y.tab.c –ll –ly –lm

$./a .out

Algorithm:

Write a program to implement YACC for Subset of C (for loop) statement.

LEX program:

1. Declare header files y.tab.h which contains information of the tokens and also declare variable yylval within %{ and %}.
2. End of declaration section with %%
3. Write the Regular Expression for: FOR, OB, CB, SM, CON, EQ, ID, NUM, INC, DEC.
4. If match found for regular expression then write action that store token in yylval where p is pointer declared in YACC and return the valve of token.
5. End rule-action section by %%
6. Subroutines section is optional.

1. Declaration of header files and set flag=0;
2. Declare tokens FOR, OB, CB, SM, CON, EQ, ID, NUM, INC, DEC.
3. End of declaration section by %%
4. State Context Free Grammar for FOR loop in rule section and write appropriate action for same.

```
S   : FOR OPBR E1 SEMIC E2 SEMIC E3 CLBR  { printf("Accepted!");flag=1; }
      ;

    | ID EQ NUM
    ;
E2 : ID RELOP ID
    | ID RELOP NUM
```

;

E3 : ID INC

| ID DEC

;

5. End the translation rule section by %%
6. Define main() function to call yyparse() function to parse an input stream
7. Define yyerror() function to displaying any error message when a yacc detects a syntax error. yyerror(const char *msg) { if(flag==0); printf("\n\n Syntax is Wrong"); }

Conclusion:

The *yacc* command accepts a language that is used to define a grammar for a target language to be parsed by the tables and code generated by *yacc*. The language accepted by *yacc* as a grammar for the target language is described below using the *yacc* input language itself.

The input *grammar* includes rules describing the input structure of the target language and code to be invoked when these rules are recognized to provide the associated semantic action. The code to be executed will appear as bodies of text that are intended to be C-language code. The C-language inclusions are presumed to form a correct function when processed by *yacc* into its output files.

FAQ's

1. For which phase of compilation is YACC used.
2. What is the role of parser? YACC is which kind of a parser?
3. How the tokens generated from LEX are passed to YACC?
4. How y. tab.h is generated? What are the contents of it?
5. Explain the grammar defined in yacc file.

Exp-03

Aim:

Generate and populate appropriate Symbol Table.

Theory:

Figure 1 depicts the schematic of a two pass language processor. The first pass performs analysis of the source program and reflects its results in the intermediate representation (IR). The second pass reads and analyses the IR to perform synthesis of the target program. This avoids repeated process sing of the source program.

The first pass is concerned exclusively with source language issues. Hence it is called the *FRONT END* of the language processor. The second pass is concerned with
the language processor.

The Front End

The front end performs

- Lexical analysis,
- Syntax analysis and
- Semantic analysis of the source program.

Each kind of analysis involves the following functions:

Determine validity of a source statement from the viewpoint of the analysis.

1. Determine the 'content' of a source statement.
2. Construct a suitable representation of the source statement for use by subsequent analysis functions, or by the synthesis phase of the language processor.

The word 'content' has different connotations in lexical, syntax and semantic analysis.

- In lexical analysis, the content is the lexical class to which each lexical unit belongs,

- In syntax analysis it is the syntactic structure of a source statement.

- In semantic analysis the content is the meaning of a statement—for a declaration statement, it is the set of attributes of a declared variable (e.g. type, length and dimensionality), while for an imperative statement, it is the sequence of actions implied by the statement.

  Each analysis represents the 'content' of a source statement in the form of

(1) Tables of information, and (2) description of the source statement. Subsequent analysis uses this information for its own purposes and either adds information to these tables and descriptions, or constructs its own tables and descriptions.

Output of the front end

The IR produced by the front end consists of two components:

1. Tables of information
2. An *INTERMEDIATE CODE* (IC) which is a description of the source program.

TABLES

Tables contain the information obtained during different analyses of SP. The most important table is the symbol table which contains information concerning all identifiers used in the SP. The symbol table is built during lexical analysis. Semantic analysis adds information concerning symbol attributes while processing declaration statements. It may also add new names designating temporary results.

INTERMEDIATE CODE (I C)

The IC is a sequence of IC units; each IC unit representing the meaning of one action in SP. IC units may contain references to the information in various tables.

Example

Figure 2 shows the IR produced by the analysis phase for the program i:

```
integer;
a,b: real;
a := b+i;
```

Symbol Table:

No.	Symbol	Type	Length	Address
1	i	int		
2	a	real		
3	b	real		
4	i*	real		
5	temp	real		

Intermediate code

1. Convert (id, #1) to real, giving (id, #4)
2. Add (id, #4) to (id, #3) giving (id, #5)
3. Store (id, #5) in (Id, #2)

Figure 2: Intermediate Representation

The symbol table contains information concerning the identifiers and their types. This information is determined during lexical and semantic analysis, respectively. In IC, the specification (Id, #1) refers to the id occupying the first entry in the table. i* and temp are temporary names added during semantic analysis of the assignment statement.

The Back End

The back end performs memory allocation and code generation.

Memory allocation

Memory allocation is a simple task given the presence of the symbol table. The memory requirement of an identifier is computed from its type, length and dimen-sionality, and memory is allocated to it. The address of the memory area is entered in the symbol table.

Example

After memory allocation, the symbol table looks as shown in Figure 3. The entries for i* and temp are not shown because memory allocation is not needed for these id's.

No.	Symbol	Type	Length	Address
1	i	int		2000
2	a	real		2001
3	b	real		2002

Figure 3: Symbol table after memory allocation

Certain decisions have to precede memory allocation, for example, whether i* and temp should be allocated memory. These decisions are taken in the preparatory steps of code generation.

Code generation

Code generation uses knowledge of the target architecture, viz. knowledge of instructions and addressing modes in the target computer, to select the appropriate instructions.

Example

For the sequence of actions for the assignment statement a :* b+i;

(i) Convert i to real, giving i*.
(ii) Add i* to b, giving temp,
(iii) Store temp in a.

the synthesis phase may decide to hold the values of i* and temp in machine registers and may generate the assembly code

```
CONV_R    AREG, I
ADD_R     AREG, B
MOVEM     AREG, A
```

where CONV_R converts the value of I into the real representation and leaves the result in AREG. ADD_R performs the addition in real mode and MOVEM puts the result into the memory area allocated to A.

Input:

```
#include<stdio.h>
void main()
{
```

```
        double a=10;
        int b=5; float
        c=2.3f; char
        d='A';
}
```

Output:

Generating the symbol table

ID	Datatype	Variable	Value
1	double	a	10
2	int	b	5
3	float	c	2.3f
4	char	d	‘A’

Conclusion:

Assg - 4

Title:- Implementat^n of semantic analysis, Semantic analysis operat^n (type-checking, Verificat^n of funct^n parameters, Variable declarat^n & (Oercions) possibly using an Attributed Translat^n grammer.

Semantics :

Semantic of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other.

Semantic analysis judges whether the syntax structure constructed in source program drives any meaning or not

CFG + semantic rules = Syntax Directed Definitions.

eg. int a = "value";

as it is lexically and structurally correct, but it should generate a semantic error as the type of assignment differs.

following tasks should be performed in semantic analysis :

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

Semantic Errors :

following are some of the semantic errors :

- Type Mismatch
- Undeclared Variable.
- Reserved identifier misuse.
- Multiple declaration of variable in scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

- **Attribute Grammar :**

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non terminals in order to provide context-sensitive information.

Each attribute has well defined domain of values, such as integer, float, character, string & expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar & it can help specify the syntax and semantics of programming language.

example : $E \rightarrow E+T$ { E.value = E.value + T.value }

the right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Values of non-terminals E & T are added together & result is copied to the non-terminal E.

• Synthesized attributes:

These attributes get values from the attribute values of their child nodes.

$S \rightarrow ABC$

If S is taking values from the attribute values of their child node)

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

as in previous ex. ($E \rightarrow E+T$), the present node E gets its value from its child nodes,

it never takes value from their parent nodes or any sibling nodes.

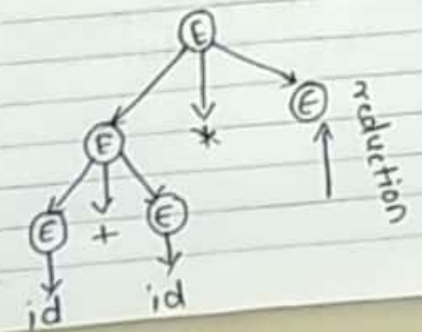
• Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and or siblings.

$S \rightarrow ABC$

expansion :-

when non-terminal is expanded to terminals as per a grammatical rule.



· **Reduction :**

When terminal is reduced to its corresponding non-terminal according to grammar rules, syntax trees are parsed top-down & left to right. whenever reduction occurs, we apply its corresponding semantic rules.

Semantic analysis uses Syntax Directed translations to perform tasks.

Semantic analyzer attaches attribute information with AST, which are called attribute AST.

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

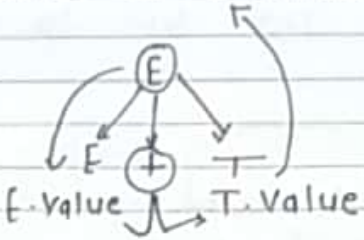
eg.

```
int value = 5;
<type, "integer">
<presentvalue, "5">
```

⊙ **S-attributed SDT :**

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

E.Value = E.value + T.Value



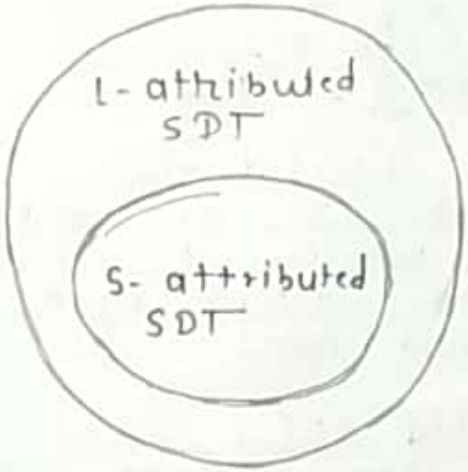
As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

- **L-attributed SDT:**
  This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.
  In L-attributed SDTs are, non-terminal can get values from it parent, child and sibling nodes.

  $S \rightarrow ABC$

  S can take values from A, B and C. A can take values from S only. B can take values from S and A, C can get values from S, A + B Non terminal can get value from sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

Conclusion :

Assignment No.	5
Title	Implement the front end of a compiler that generates the three address code for a simple language.
Roll No.	
Class	B.E. (C.E.)
Date	
Subject	Laboratory Practice-IV
Signature	

Assignment No: 5

Title: Implement the front end of a compiler that generates the three address code for a simple language.

Aim: Write an attributed translation grammar to recognize declarations of simple variables, "for", assignment, if, if-else statements as per syntax of C or Pascal and generate equivalent three address code for the given input made up of constructs mentioned above using LEX and YACC. Write a code to store the identifiers from the input in a symbol table and also to record other relevant information about the identifiers. Display all records stored in the symbol table.

Objective: To learn the function of compiler by:

- To understand fourth phase of compiler: Intermediate code generation.
- To learn and use compiler writing tools.
- To learn how to write three address code for given statement.

Theory:

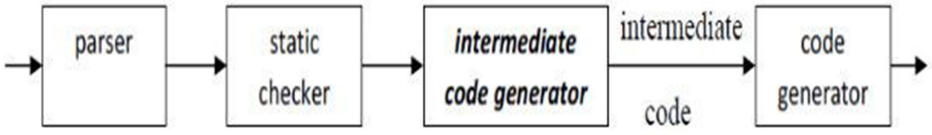
Introduction:

In the analysis - synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. Ideally, details of the source language are confined to the front end, and details of the target machine to the back end. The front end translates a source program into an intermediate representation from which the back end generates target code. With a suitably defined intermediate representation, a compiler for language i and machine j can then be built by combining the front end for language i with the back end for machine j. This approach to creating suite of compilers can save a considerable amount of effort: m x n compilers can be built by writing just m front ends and n back ends.

Benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation

Position of intermediate code generator



Intermediate Languages:

Three ways of intermediate representation:

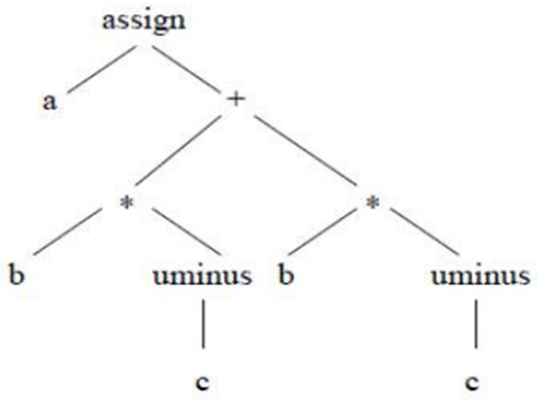
- Syntax tree
- Postfix notation
- Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

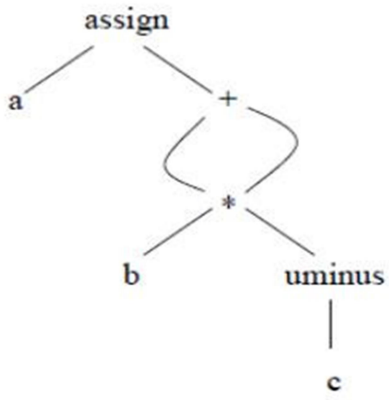
Graphical Representations:

1. ***Syntax tree:***

   A syntax tree depicts the natural hierarchical structure of a source program. A dag(Directed Acyclic Graph) gives the same information but in a more compact way because common sub expressions are identified. A syntax tree and dag for the assignment statement a : =b * -c + b * -c are as follows:



(a) Syntax tree



(b) Dag

2. ***Postfix notation:***

   Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

   a b c uminus * b c uminus * + assign

3. ***Three-Address Code:***

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

   form x : = y op z

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

t1 : = y * z

t2 : = x +t1

Where t1 and t2 are compiler-generated temporary names.

The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

Advantages of three-address code:

- The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- The use of names for the intermediate values computed by a program allows three address code to be easily rearranged – unlike postfix notation.

Three-address code is a liberalized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three address statements.

Types of Three-Address Statements:

The common three-address statements are:

- Assignment statements of the form x : = y op z, where op is a binary arithmetic or logical operation.
- Assignment instructions of the form x : = op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
- Copy statements of the form x : = y where the value of y is assigned to x.
- The unconditional jump goto L. The three-address statement with label L is the next to be executed.
- Conditional jumps such as if x relop y goto L. This instruction applies a relational operator (<, =, >=, etc.) to x and y, and executes the statement with label L next if x stands in relation relop to

y. If not, the three-address statement following if x relop y goto L is executed next, as in the usual sequence.

- param x and call p, n for procedure calls and return y, where y representing a returned value is optional. For example,

  param x1

  param x2

  . . . . . .

  param xn

  call p,n

  generated as part of a call of the procedure p(x1, x2, …. ,xn).

- Indexed assignments of the form x : = y[i] and x[i] : = y.

- Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are: Quadruples, Triples, Indirect triples.

A. Quadruples:

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

- The op field contains an internal code for the operator. The 3 address statement x = y op z is represented by placing y in arg1, z in arg2 and x in result.

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

- Fig a) shows quadruples for the assignment a : b * c + b * c

B. Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields: op, arg1 and arg2.
- The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as triples.
- Fig b) shows the triples for the assignment statement a: = b * c + b * c.

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	: =	t5		a

Fig a): Quadruples

	op	Arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Fig b): Triple

Quadruples & Triple representation of three-address statement

C. Indirect triples:

- Indirect triple representation is the listing pointers to triples rather-than listing the triples themselves.
- Let us use an array statement to list pointers to triples in the desired order.
- Fig c) shows the indirect triple representation.

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Fig c): Indirect triples representation of three address statements

Steps to execute the program

$ lex filename.l (eg: comp.l)

$ yacc -d filename.y (eg: comp.y)

$cc lex.yy.c y.tab.c –ll –ly -lm

$./a .out

Algorithm:

Write a LEX and YACC program to generate Intermediate Code for arithmetic expression

LEX program:

1. Declaration of header files specially y.tab.h which contains declaration for Letter, Digit, expr.
2. End declaration section by %%
3. Match regular expression.
4. If match found then convert it into char and store it in yylval.p where p is pointer declared in YACC
5. Return token
6. If input contains new line character (\n) then return 0
7. If input contains „.“ then return yytext[0]
8. End rule-action section by %%
9. Declare main function
   a. open file given at command line

b. if any error occurs then print error and exit

c. assign file pointer fp to yyin

d. call function yylex until file ends

10. End

1. Declaration of header files
2. Declare structure for three address code representation having fields of argument1, argument2, operator, result.
3. Declare pointer of char type in union.
4. Declare token expr of type pointer p.
5. Give precedence to „*“,“/“.
6. Give precedence to „+“,“-“.
7. End of declaration section by %%.
8. If final expression evaluates then add it to the table of three address code.
9. If input type is expression of the form.
    a. exp“+“exp then add to table the argument1, argument2, operator.
    b. exp“-“exp then add to table the argument1, argument2, operator.
    c. exp“*“exp then add to table the argument1, argument2, operator.
    d. exp“/“exp then add to table the argument1, argument2, operator.
    e. (exp) then assign $2 to $$.
    f. Digit OR Letter then assigns $1 to $$.
10. End the section by %%.
11. Declare file *yyin externally.
12. Declare main function and call yyparse function untill yyin ends
13. Declare yyerror for if any error occurs.
14. Declare char pointer s to print error.
15. Print error message.

16. End of the program.

Addtotable function will add the argument1, argument2, operator and temporary variable to the structure array of three address code.

Three address code function will print the values from the structure in the form first temporary variable, argument1, operator, argument2

Quadruple Function will print the values from the structure in the form first operator, argument1, argument2, result field

Triple Function will print the values from the structure in the form first argument1, argument2, and operator. The temporary variables in this form are integer / index instead of variables.

Conclusion:

Thus we have studied how to generate intermediate code. A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are: Quadruples, Triples, Indirect triples that we have studied.

FAQ's

1. What are the different forms of ICG?
2. What are the difference between syntax tree and DAG?
3. What are advantages of 3-address code?
4. Which representation of 3-address code is better than other and why? Justify.
5. What is role of Intermediate code in compiler?

Assg - 6

Title :- A Register Allocation algorithm that translates the given code into one with a fixed number of registers.

Theory :

Register Allocation : In compiler optimization, Register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers.
Register Allocation can happen over a basic block (local register allocation), over a whole function / procedure (global register allocation), or across function boundaries traversed via call-graph (interprocedural register allocation).

Principle : In many programming languages, the programmer may use any number of variables. The computer can quickly read & write registers in the CPU, so the computer program runs faster when more variables can be in the CPU's registers. Also, sometimes code accessing registers is more compact, so the code is smaller & can be fetched faster if it uses registers rather than memory.

- In Three Address Code, there are unlimited no of variables.

These are two important activities done while using Registers:

1) Register Allocation: During register allocation, select appropriate set of variables that will reside in registers.

2) Register assignment: During register assignment, pick up the specific register in which corresponding variable will reside.

Obtaining the optimal (minimum) assignment of registers to variable is difficult.

Certain machine require register pairs such as even odd numbered registered for some operands and results.

For eg in IBM systems integer multiplication requires register pair.

Consider the 3 address code:

$t_1 := a + b$

$t_1 := t_1 / c$

$t_1 := t_1 / d$

The efficient machine code sequence will be

```
mov a,R0
add b,R0
mul c,R0
div d,R0
mov R0,t1
```

- Various Strategies used in register allocation and assignment :-

1. Global register Allocation
2. Usage Count
3. Register assignment for outer loop.
4. Graph coloring for register assignment.

1) Global Register Allocation :-

While generating the code the registers are used to hold the values for the duration of single block. The allocation of variables to specific registers that is consistent across the block boundaries is called Global Register allocation.

Strategies adopted while doing the global register allocation :-

- The global register allocation has a strategy of storing the most frequently used variables in fixed registers throughout the loop.
- Another strategy is to assign some fixed no. of global register to hold the most active values in each inner loop.
- The registers not already allocated may be used to hold values local to one block.
- In certain languages like C or Bliss programmer can do the register allocat" by using register allocation.

2) Usage count:

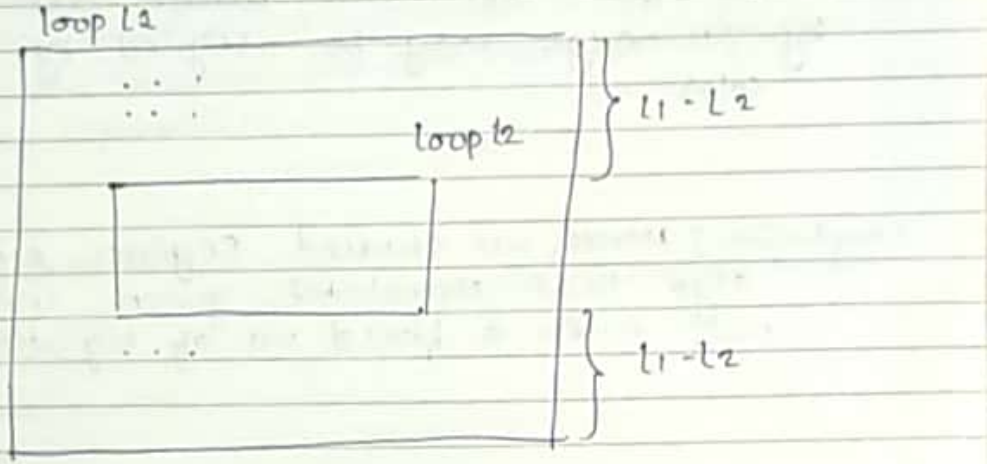
The usage count is the count for the use of some variable 'x' in some register used in any basic block. The usage count gives the idea about how many units of cost can be saved by selecting a specific variable for global register allocation. The approximate formula for usage count for the loop 'L' is some basic block 'B' is given as

$$\sum_{\text{block } B \text{ in } L} (use(x,B) + 2 * live(x,B))$$

where use(x,B) is no of times x used in block B prior to any defn of x and live(x,B) = 1 if x is live on exit from B; otherwise live(x) = 0.

3) Register assignment for outer loop:

Consider there are two loops L1 (outer loop) and L2 (inner loop). and allocation of variable a is to be done to some register.



Criteria for register assignment for outer loop:-

1) If a is allocated in loop L2, then it should not be allocated in L1 - L2.
2) If a is allocated in L1 & not allocated in L2 then store a on a entrance to L2 and load a while leaving L2.
3) If a is allocated in L2 and not in L1 then load a on entrance of L2 and store a on exit from L2.

4) Graph coloring for Register Assignment:-
It is relatively simple method which can be used for some of the scheduling problems.
For this, we first need to construct an interference graph.
Then we color the interference graph using k different colours, where k is the no of registers available for allocation.

No pair of nodes which are connected by an edge may be assigned by the same color.

Conclusion :- Hence, we studied Register Allocation algo that translates given code into one with a fixed no of registers.

EXPERIMENT NO: 07

1. Title:

Implementation of Instruction Scheduling Algorithm.

2. Objectives:

- To understand OS & SCHEDULLING Concepts
- To implement Scheduling FCFS, SJF, RR & Priority algorithms
- To study about Scheduling and scheduler

3. Outcomes:

After completion of this assignment students will be able to:

- Knowledge Scheduling policies
- Compare different scheduling algorithms

Theory Concepts:

CPU Scheduling:

• CPU scheduling refers to a set of policies and mechanisms built into the operating systems that govern the order in which the work to be done by a computer system is completed.
• Scheduler is an OS module that selects the next job to be admitted into the system and next process to run.
• The primary objective of scheduling is to optimize system performance in accordance with the criteria deemed most important by the system designers.

What is scheduling?

Scheduling is defined as the process that governs the order in which the work is to be done. Scheduling is done in the areas where more no. of jobs or works are to be performed. Then it requires some plan i.e. scheduling that means how the jobs are to be performed i.e. order. CPU scheduling is best example of scheduling.

What is scheduler?

1. Scheduler in an OS module that selects the next job to be admitted into the system and the next process to run.
2. Primary objective of the scheduler is to optimize system performance in accordance with the criteria deemed by the system designers. In short, scheduler is that module of OS which schedules the programs in an efficient manner.

Necessity of scheduling

- Scheduling is required when no. of jobs are to be performed by CPU.
- Scheduling provides mechanism to give order to each work to be done.
- Primary objective of scheduling is to optimize system performance.
- Scheduling provides the ease to CPU to execute the processes in efficient manner.

Types of schedulers

In general, there are three different types of schedulers which may co-exist in a complex operating system.
- Long term scheduler
- Medium term scheduler
- Short term scheduler.

Long Term Scheduler

- The long term scheduler, when present works with the batch queue and selects the next batch job to be executed.
- Batch is usually reserved for resource intensive (processor time, memory, special I/O devices) low priority programs that may be used fillers of low activity of interactive jobs.
- Batch jobs usually also contains programmer-assigned or system-assigned estimates of their resource needs such as memory size, expected execution time and device requirements.
- Primary goal of long term scheduler is to provide a balanced mix of jobs.

Medium Term Scheduler

- After executing for a while, a running process may because suspended by making an I/O request or by issuing a system call.
- When number of processes becomes suspended, the remaining supply of ready processes in systems where all suspended processes remains resident in memory may become reduced to a level that impairs functioning of schedulers.

• The medium term scheduler is in charge of handling the swapped out processes.
• It has little to do while a process is remained as suspended.

Short Term Scheduler

• The short term scheduler allocates the processor among the pool of ready processes resident in the memory.
• Its main objective is to maximize system performance in accordance with the chosen set of criteria.
• Some of the events introduced thus for that cause rescheduling by virtue of their ability to change the global system state are:
• Clock ticks
• Interrupt and I/O completions
• Most operational OS calls
• Sending and receiving of signals
• Activation of interactive programs.
• Whenever one of these events occurs ,the OS involves the short term scheduler.

Scheduling Criteria :

- CPU Utilization:

Keep the CPU as busy as possible. It range from 0 to 100%. In practice, it range from 40 to 90%.

- Throughput:

Throughput is the rate at which processes are completed per unit of time.

- Turnaround time:

This is the how long a process takes to execute a process. It is calculated as the time gap between the submission of a process and its completion.

- Waiting time:

Waiting time is the sum of the time periods spent in waiting in the ready queue.

- Response time:

Response time is the time it takes to start responding from submission time. It is calculated as the amount of time it takes from when a request was submitted until the first response is produced.

Non-preemptive Scheduling:

In non-preemptive mode, once if a process enters into running state, it continues to execute until it terminates or blocks itself to wait for Input/output or by requesting some operating system service.

Preemptive Scheduling:

In preemptive mode, currently running process may be interrupted and moved to the ready State by the operating system.
When a new process arrives or when an interrupt occurs, preemptive policies may incur greater overhead than non-preemptive version but preemptive version may provide better service.

It is desirable to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time and response time.

Types of scheduling Algorithms

• In general, scheduling disciplines may be pre-emptive or non-pre-emptive .
• In batch, non-pre-emptive implies that once scheduled, a selected job turns to completion.
There are different types of scheduling algorithms such as:

- FCFS(First Come First Serve)
- SJF(Short Job First)
- Priority scheduling
- Round Robin scheduling algorithm

First Come First Serve Algorithm

• FCFS is working on the simplest scheduling discipline.
• The workload is simply processed in an order of their arrival, with no pre-emption.
• FCFS scheduling may result into poor performance.
• Since there is no discrimination on the basis of required services, short jobs may considerable in turn around delay and waiting time.

Advantages

• Better for long processes

• Simple method (i.e., minimum overhead on processor)
• No starvation

Disadvantages

• Convoy effect occurs. Even very small process should wait for its turn to come to utilize the CPU. Short process behind long process results in lower CPU utilization.
• Throughput is not emphasized.

First Come, First Served

Process	Burst Time
P1	24
P2	3
P3	3

- Suppose that the processes arrive in the order: *P1* , *P2* , *P3*
- The Gantt Chart for the schedule is:

- Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Note : solve complete e.g. as we studied in practical(above is just sample e.g.). you can take any e.g.

Shortest Job First Algorithm:

- his is also known as shortest job first, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

Advantages

• It gives superior turnaround time performance to shortest process next because a short job is given immediate preference to a running longer job.
• Throughput is high.

Disadvantages

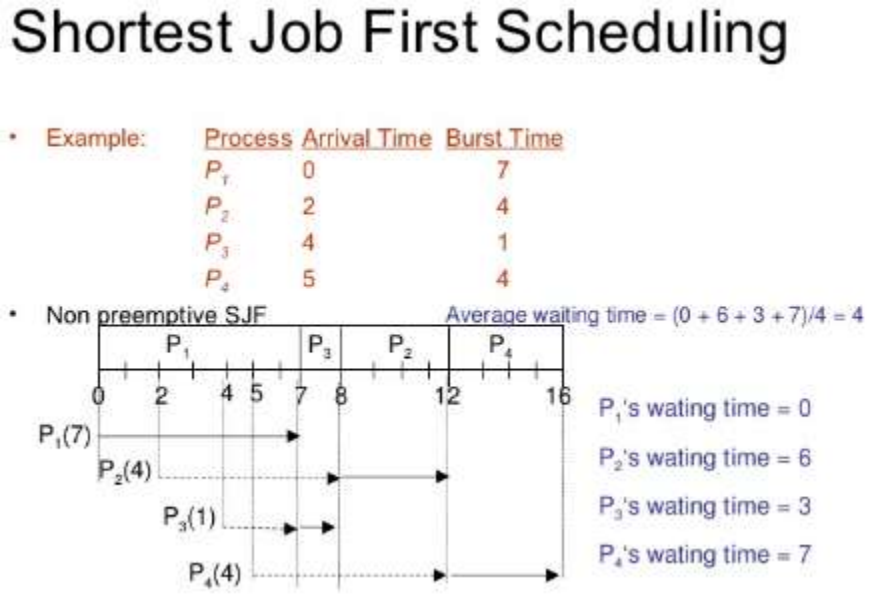
• Elapsed time (i.e., execution-completed-time) must be recorded, it results an additional overhead on the processor.
• Starvation may be possible for the longer processes.

This algorithm is divided into two types:

• Pre-emptive SJF
• Non-pre-emptive SJF

• Pre-emptive SJF Algorithm:

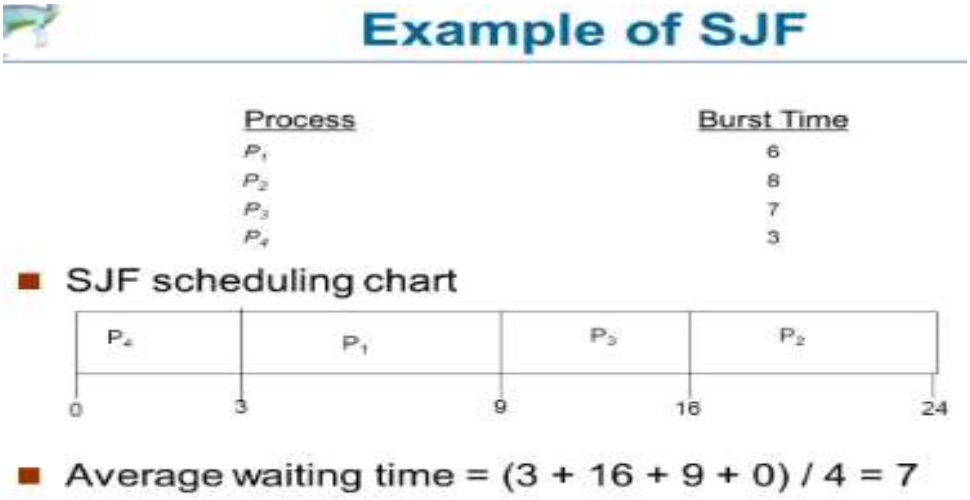
In this type of SJF, the shortest job is executed 1st. the job having least arrival time is taken first for execution. It is executed till the next job arrival is reached.



Note: solve complete e.g. as we studied in practical(above is just sample e.g.). you can take any e.g.

Non-pre-emptive SJF Algorithm:

In this algorithm, job having less burst time is selected 1st for execution. It is executed for its total burst time and then the next job having least burst time is selected.



Note: solve complete e.g. as we studied in practical(above is just sample e.g.). you can take any e.g.

Round Robin Scheduling:

• Round Robin is the preemptive process scheduling algorithm.
• Each process is provided a fix time to execute, it is called a quantum.
• Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
• Context switching is used to save states of preempted processed

Advantages

• Round-robin is effective in a general-purpose, times-sharing system or transaction-processing system.

• Fair treatment for all the processes.
• Overhead on processor is low.
• Overhead on processor is low.
• Good response time for short processes.

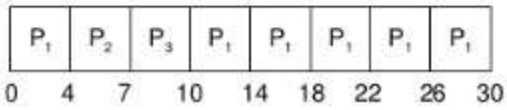
Disadvantages

• Care must be taken in choosing quantum value.
• Processing overhead is there in handling clock interrupt.
• Throughput is low if time quantum is too small.

Round Robin

Process	Burst Time
P1	24
P2	3
P3	3

- Quantum time = 4 milliseconds
- The Gantt chart is:



- Average waiting time = {[0+(10-4)]+4+7}/3 = 5.6

Note : solve complete e.g. as we studied in practical(above is just sample e.g.). you can take any e.g.

Priority Scheduling:

• Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
• Each process is assigned a priority. Process with highest priority is to be executed first and so on.
• Processes with same priority are executed on first come first served basis.
• Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Advantage

• Good response for the highest priority processes.

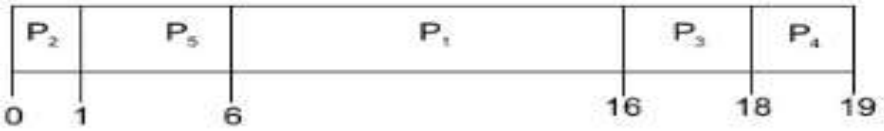
Disadvantage

• Starvation may be possible for the lowest priority processes.

Priority

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

- **Gantt Chart**



- **Average waiting time = (6 + 0 + 16 + 18 + 1)/5 = 8.2**

Note: solve complete e.g. as we studied in practical (above is just sample e.g.). You can take any e.g.

Conclusion:

Title :- Implement Local and Global Code Optimization such as Common Sub-Expression Elimination, Copy Propogation Dead-Code Elimination, Loop and Basic Block Optimization (Optional).

Theory :-

~~Code~~ Optimization

- Optimization seek to improve a programs utilization of some resource.
  - Execution time
  - Code size
  - Network messages sent
  - Battery powered used, etc.
- Optimization should not alter what the program computes the answer must be the same
- Observable behaviour must be the same.

Clamification of Optimization :-

1) Local Optimization
2) Global Optimization

* Local Optimization :-

- local Optimization phase is also called Optional phase and is needed only to make the object program more efficient.
- It involves examing sequence of instruction put out by the code generator to find unnecessary or redundent instructions.
- local Optimization is also called as Machine-dependent Optimization.
- In this Optimization, the compiler takes in the intermediate

code and transform a part of the code that does not involve any CPU registers and/or absolute memory allocations.

For Example:

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

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

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

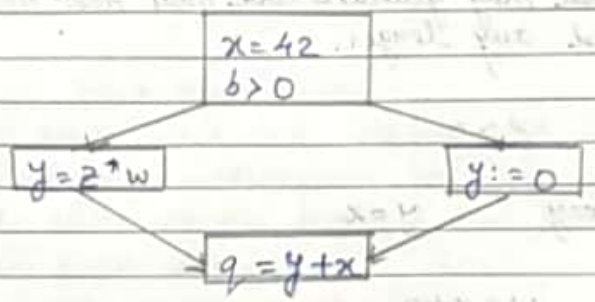
should not only save the CPU cycles but can be used on any processor.

Global Optimization:-

- It is also called as machine independent Optimization or Unreachable Code Elimination.
- Removing unreachable code makes the program smaller due to instruction cache effects.
- The local Optimization has very restricted scope on the other hand the global Optimization is applied a broad scope such as procedure or function body.
- Global Optimization is a program is represented in the form of program flow graph.

- There are two types of analysis performed for global optimization :
  - Control flow analysis
  - Data flow analysis

Control flow graph :-



* Common Sub-Expression Elimination :-
A common subexpression is an expression that appears multiple times on a right-hand-side context where the operands have the same value in each case so that the expression will yield the same value.

Assume :-
- - A basic block is in single form assignment form.
- A definition x:= is the first use of x in a block
- All assignments with same RHS compute the same value.

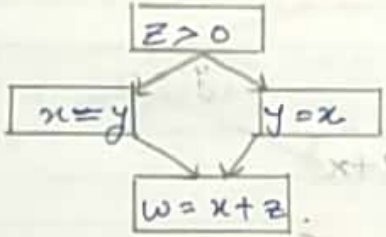
Example :-

x := y+z ⇒ x := y+z
....
w := y+z w = x

(The values of x, y and z do not change in the code)

Copy Propogation :-

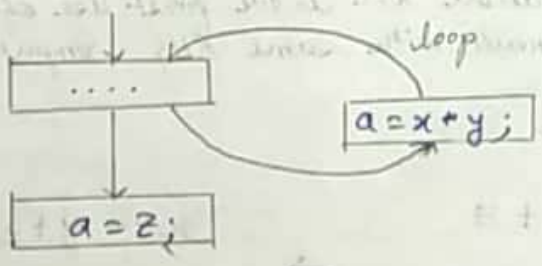
- Data flow analysis to determine which instruction are candidates for global copy propogation.
- On assignment $x = y$, replace later uses of x with use y, provided there are no intervening assignment to x or y.
- Copy propogation may generate code that does not need to be evaluated any longer.



Dead code Elimination :-

Dead code is one or more than one code statements, which are — either never executed or unreachable
— Or if executed, their output is never used.
Thus, dead code play no role in any program operation and therefore it can simply be eliminated.

Partially dead code.



If $w := RHS$ appears in a block.
w does not appear anywhere else in the program

Then the statement w := RHS is dead and can be eliminated.

Example :- (a is not used anywhere else).

```
x = z+y        b := z+y
a : x    ⇒     a := b    ⇒    b := z+y
x := 2*x       x := 2+b       x := 2+b
```

* Basic-Block Optimization :-

A basic block is a sequence of consecutive intermediate language statements in which flow of control can only enter at the beginning and leave at the end.

- Basic block starts.
  - first statement of program.
  - Statement that are targeted of any branch
  - Statement that follow branch statement.

```
w = 0;
x = x+y;
y = 0;
if (x > z)
{
  y = x
  x++;
}
else
{
  y = z;
  z++;
}
w = x+z;
```

Source code

```
w = 0;
x = x+y;
y = 0;
if (x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

```
w = x+z;
```

Basic Blocks

Loop Optimization :-

Loop Optimization is most valuable machine-independent Optimization because program's inner loop taker bulk to time of a programmer.

Techniques are as follow:-

1) Code motion
2) Induction - Variable elimination
3) Strength reduction.

B1

```
w=0
x=x+y
y=0;
if(x>z)
```

B2

```
y=z;
x++;
```

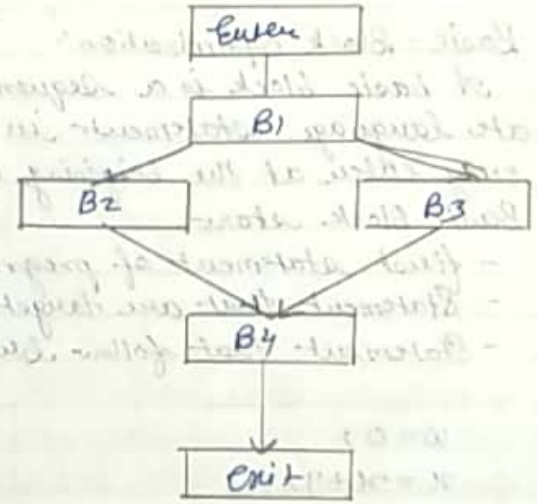
B3

```
y=z;
z++;
```

B4

```
w=x+z;
```

Basic Block.



flow graph of loop Optimization

Conclusion:-

Hence we implement local, global and Loop and Basic Optimization.

A	P	J	T	Sign
(5)	(4)	(3)	(10)	