UNIT I

CHAPTER 1

Introduction to System Programming

Syllabus Topics

Components of System Software, Language Processing Activities, Fundaments of Language Processing.

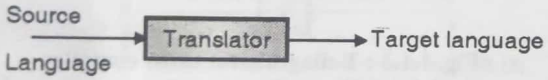
1.1 Introduction

Language processing activity involves translation of a program written in a high level language into machine code. A translator permits the programmer to express his algorithm in a language other than that of a machine. Some of the benefits of writing a program in a language other than a machine language are as follows :

(1) Increase in the programmer's productivity – It is easy to write a program in a high level language.

(2) Machine independence – A program written in a high level language is machine independent.

The input to a translator is a program written in a source language (high level language). The output of a translator is a program in target language (machine code).



(S1.1)**Fig. 1.1.1 : A translator**

There is a difference between source language and the target language.

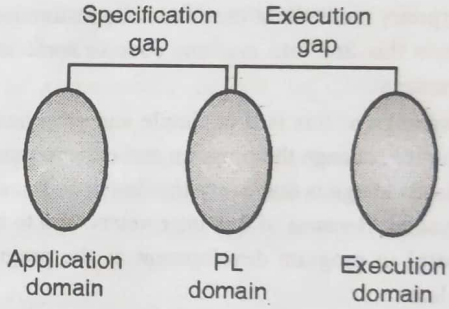
- A source language is closure to application domain. A source language should provide features necessary for expressing an algorithm.
- The ideas expressed in source domain must ultimately be interpreted in terms of execution domain of the computer system.
- Semantic gap between the two domains often implies efforts needed by a translator to convert a program written in source language into machine code. If the semantic gap is less then it is easy write a translator. But this will have adverse consequences in program development, particularly large development times, large development efforts and poor quality of software.

The development of large programs involve two steps :

(1) Specification, design and writing of algorithm.

(2) Implementation of algorithm through a programming language (PL).

Software development using a programming language introduces a new domain, the PL domain. It is shown in Fig. 1.1.2.



(S1.2)**Fig. 1.1.2 : Specification and execution gap**

- Specification gap can be reduced by selecting a programming language which provides features suited for writing a particular application; but this will increase the execution gap.

☞ Definition of a language processor

A language processor is software which bridges a specification or execution gap. There are different types of language processers :

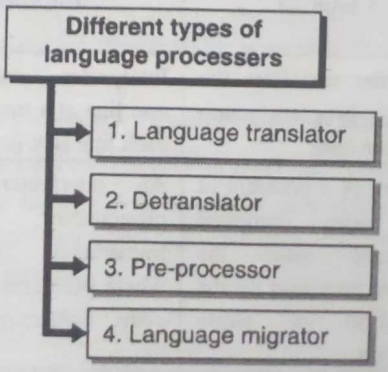


Fig. C1.1 : Types of language processor

Scanned with CamScanner

→ (1) **A language translator :** It translates a program written in any programming language into machine code. A C-Compiler is a language translator for a program written in C-Language.

→ (2) **A detranslator** does the reverse of a language translator.

→ (3) **A preprocessor** bridges an execution gap but it is not a language translator.

→ (4) **A language migrator** bridges the specification gaps between two programming languages.

1.1.1 An Interpreter → (Dec. 2013)

Q. What is interpreter ? Explain the role of interpreter with suitable example. **SPPU - Dec. 2013, 8 Marks**

An interpreter is useful during program development. A program execution requires either a compiler or interpreter. A compiler translates a source program into machine instructions. These machine instructions can be executed by a computer. In case of an interpreter, the interpreter accepts the source program and performs the actions associated with the instructions. Thus it creates a virtual execution environment. The interpreter has the control of the execution of the program during execution.

An interpreter reads the source code one instruction or line at a time, converts this line into machine code or some intermediate form and executes it.

The advantage of this is it is simple and you can interrupt it while it is running, change the program and either continue or start again. The disadvantage is that every line has to be translated every time it is executed. Because of this interpreters tend to be slow but it is very useful in program development as the program can be modified on line.

1.1.2 Difference between Interpreter and Compiler

→ (Aug. 2015, Oct. 2016, May 2017)

Q. Compare Compiler and Interpreter. **SPPU - Aug. 2015(In Sem), May 2017, 4 Marks**

Q. Explain Compilers and Interpreters. **SPPU - Oct. 2016(In Sem), 6 Marks**

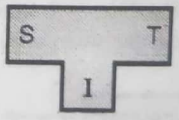
Sr. No.	Compiler	Interpreter
1.	Compiler translates the entire program into machine code.	Interpreter translates code one line at a time, executing each line as it is translated.
2.	If there is a problem in the code, compiled programs make the programmer wait for the execution of entire program.	An interpreter lets the programmer know immediately when and where problems exist in the code.
3.	Compilers produce better optimised code that generally runs faster and compiled code is self sufficient and can be run on their intended platforms without the compiler present.	Interpreters can be easier to use and produce more immediate results. However the source code of an interpreted language cannot run without the interpreter. Program execution is relatively slow due to interpretation overhead.
4.	A compiler spends a lot of time analyzing and generating machine code. It is not suited during program development.	Relatively little time is spent in analyzing as it executes line by line.

1.1.3 A Cross Compiler

A cross compiler is a type of compiler. It creates an executable code for a platform other than the one on which the compiler is run. In short, we can say the cross compiler runs on machine x and generates machine code for machine y.

- Cross compilers are useful for microcontrollers that do not support an operating system.
- Cross compiler is used to compile for a platform upon which it is not feasible to do the compiling.
- The fundamental use of a cross compiler is to separate the build environment form target environment. It is useful for embedded computers.

A T-diagram for cross-compiler is shown in Fig. 1.1.3.



(S1.3)**Fig. 1.1.3 : T-diagram for cross compiler**

A compiler is characterized by three languages :

(1) The source language that it compiles.
(2) The target language T that it generates.
(3) The implementation language I that is used for writing the compiler.

A cross compiler runs on one machine and produces target code for another machine.

1.1.4 A Bootstrap Compiler

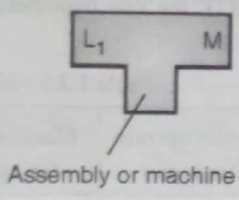
Bootstrapping is a term used for describing a technique involving writing a compiler in the target programming language which it is intended to compile.

General form of bootstrapping builds up a compiler for larger and larger sub-sets of a language.

- Suppose a new language L is to be implemented on machine M.

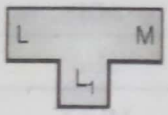
Scanned with CamScanner

- As a first step we might write a small compiler either in assembly or machine code. This compiler translates a subset L_1 of L into the target code of M.



(S1.4)**Fig. 1.1.4**

- Now, we have a working compiler for a subset of L. We can use the subset L_1 to write a compiler for L or it can be iterated a number of times.



(S1.5)**Fig. 1.1.5**

Syllabus Topic : Components of System Software

1.2 System Software

➔ **(Oct. 2016, Dec. 2016)**

Q. Explain program generation activity.

SPPU - Oct. 2016(In Sem), 4 Marks

Q. Explain the steps in program development.

SPPU - Dec. 2016, 7 Marks

System software is needed for program development. System programs were developed to make computers better adapted to the needs of their users. Some of the common system programs are :

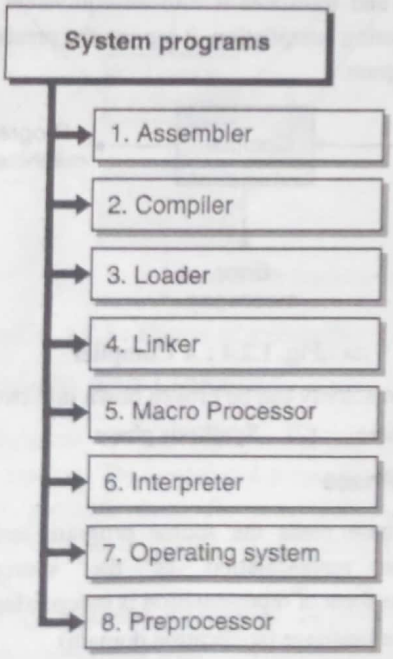


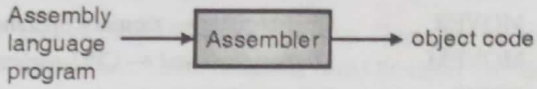
Fig. C1.2 : System programs

➔ 1.2.1 Assembler

Machine and assembly languages are low level languages since the coding for a problem is at the individual instruction level. An assembly language is machine dependent. It differs from computer to computer.

Writing a program in assembly language is more convenient than in machine language. Instead of binary sequence, as in machine language, it is written in the form of symbolic instructions.

- An assembly program is written using symbols (Mnemonic).
- An assembly program is more readable. It is difficult to understand and develop a program using machine language.
- Assembly language is machine dependent.
- An assembly program is translated into machine code before execution.
- An assembler is a translator which takes its input in the form of an assembly language program and produces machine language code as its output.



(S2.1)**Fig. 1.2.1 : Assembler**

☞ Elements of assembly language

An assembly language provides three basic features :

1. Operation codes in the form of mnemonics.
2. Symbolic operands.
3. Data declaration.

Let us consider an assembly instruction.

MOV AX, X

- **MOV** is a mnemonic opcode for the operation to be performed.
- **AX** is register operand in a symbolic form
- **X** is a memory operand in a symbolic form.

Let us consider an assembly instruction for data declaration.

X db 5

- Name of the variable is X
- X requires 1 byte of memory
- Initial value of X is set to 5.

The storage specifier **db** reserves one byte of storage for holding a value of X.

☞ Statement format

An instruction in assembly program has the following format :

1. **Label :** It is optional, if present it occupies the first field.
2. **Opcode :** It contains the symbolic operation code or storage specification.
3. **Operand :** Operand is a symbolic name for CPU-register or memory variables. There can be a number of operands in an instruction.

[Label] < opcode > < operand > [,< operand >···]

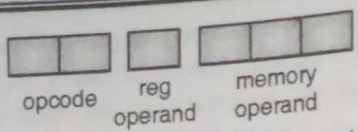
(S2.2)**Fig. 1.2.2 : Format of an assembly instruction**

A general format of an assembly instruction is shown in Fig. 1.2.2. The notation [...] indicates that the enclosed specification is optional.

☞ Operation codes

We will assume a hypothetical assembly language with the machine instruction format shown in Fig. 1.2.3.

Scanned with CamScanner



(S2.3)**Fig. 1.2.3 : Instruction format**

Our machine has three CPU registers :

1. AREG
2. BREG
3. CREG

Our machine supports 11 different operations :

1. STOP – to stop execution
2. ADD – operand1 ← operand1 + operand2
3. SUB – operand1 ← operand1 – operand2
4. MULT – operand1 ← operand1 * operand2
5. MOVER – CPU-register ← memory operand
6. MOVEM – Memory operand ← CPU-register
7. COMP – Set condition code, these condition codes will be used by the conditional branch instruction **BC**.
8. BC – Branch on condition. Conditions to be used for branching are :
   a) EQ – equal
   b) NE – not equal
   c) LT – less than
   d) GT – greater than
   e) LE – less or equal
   f) GE – greater or equal
   g) ANY
9. DIV – operand1 ← operand1 / operand2
10. READ – operand2 ← input value
11. PRINT – output ← operand2

- First operand is always a CPU register
- Second operand is always a memory operand
- READ and PRINT instructions do not use the first operand
- The stop instruction has no operand.

Example

Meaning of some typical instructions are give below :

Statement	Meaning
ADD AREG, X	AREG ←AREG + X
MOVER BREG, X	BREG ← X
MOVEM CREG, X	X ← CREG
COMP AREG, Y	Compare AREG and the memory variable Y and set the necessary condition codes. These condition codes are required by **BC** instruction.

Each symbolic opcode is associated with machine opcode. These details are stored in machine opcode table (MOT).

A MOT contains :

1. Opcode in mnemonic form
2. Machine code associated with the opcode.

MOT for our hypothetical assembly language is given in Table 1.2.1.

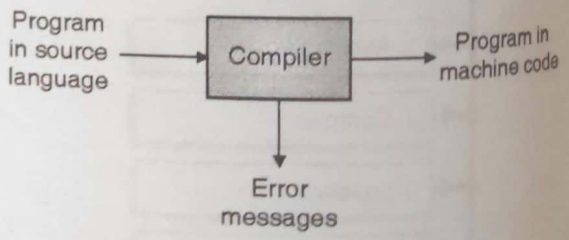
Table 1.2.1 : Machine opcode table

Symbolic opcode (mnemonic)	Machine code for opcode	Size of instruction (in number of words)
STOP	00	1
ADD	01	1
SUB	02	1
MULT	03	1
MOVER	04	1
MOVEM	05	1
COMP	06	1
BC	07	1
DIV	08	1
READ	09	1
PRINT	10	1

- Machine opcode is a two digit code.
- Size of each instruction is assumed to be of 1 word.

➜ 1.2.2 Compilers

A compiler is a program that reads a program written in source language and translates it into an equivalent program in machine code. During compilation, it reports the presence of errors in the source program.



(S4.1)**Fig. 1.2.4 : A Compiler**

Compilation activity can be broken down into two parts :

(1) Analysis phase (2) Synthesis phase

(1) Analysis Phase

Analysis phase reads the source program and creates an intermediate representation of the source program. Intermediate form of representation is independent of both :

(1) Source language (application domain)
(2) Machine language (execution domain)

(2) Synthesis Phases

The synthesis phase generates the desired program in machine code from the intermediate representation.

- Analysis phase is dependent on the source language.
- Synthesis phase is dependent on the machine for which the target code is to be generated.

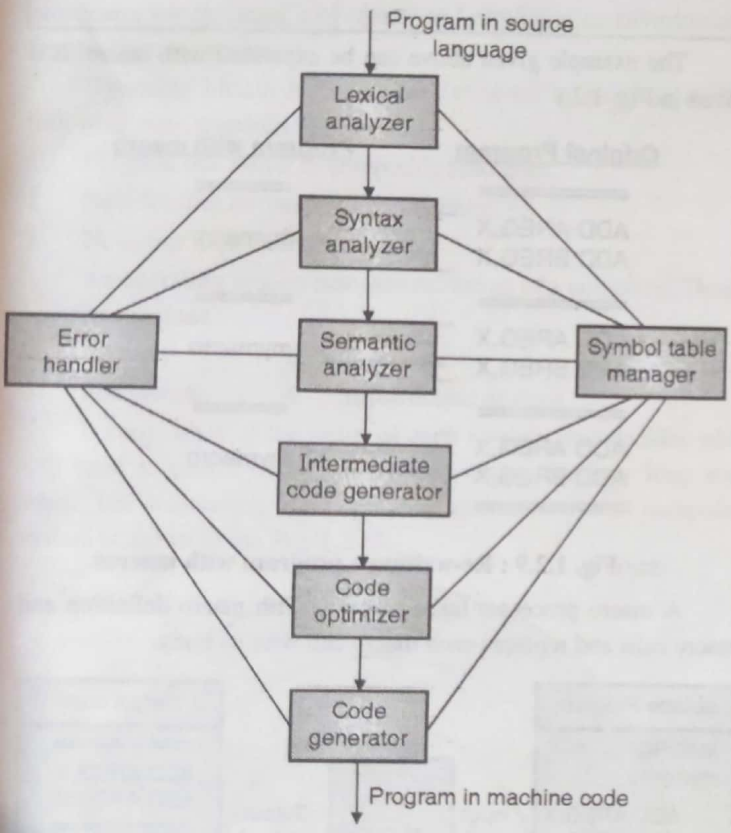
Scanned with CamScanner

Phase of compilers

A compiler works in phases. Output of one phase becomes the input of next phase. Phases of compilers are :

(1) Lexical analysis
(2) Syntax analysis
(3) Semantic analysis
(4) Intermediate code generation
(5) Code optimization
(6) Code generation.

A Typical decomposition of a compiler is shown in Fig 1.2.5.



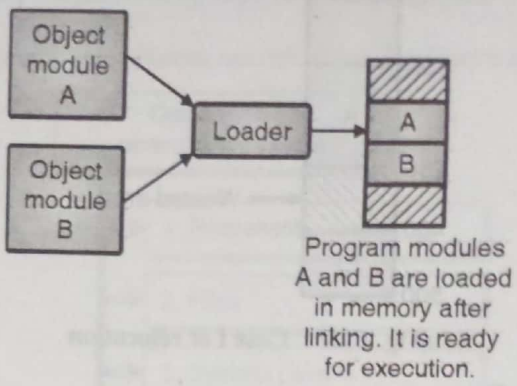
(S4.2)**Fig. 1.2.5 : Phases of a compiler**

1.2.3 Loaders and Linkers

A source program is converted to object program by assemblers and compilers. The loader is a program which accepts object codes and prepares them for execution and initiates execution. As many as four functions are performed by a loader. These functions are :

- **Allocation** of space in main memory for the programs.
- **Linking** of object modules with each other. Linking involves resolving of symbolic references between object modules.
- Adjust all address dependent locations, such as address constants, to correspond to the allocated space. It is also called **relocation.**
- Physically **loading** the machine instructions and data into the main memory.

A general loading scheme is shown in the Fig. 1.2.6.



(S5.1)**Fig. 1.2.6 : General loading scheme**

What is linking ?

Any usable program written in any language has to use functions/ subroutines. These functions could be either user defined functions or they can be library functions.

For example, consider a program written in C-language. Such a program may contain calls to functions like printf() and scanf(). During program execution the main program as well as functions must reside in the main memory. In addition, every time a function is called, the control should get transferred to the appropriate function.

- The linking process makes address of modules known to each other so that transfer of control takes place during execution.
- Passing of parameters is handled by the linker. Parameters can be passed by value or by reference. A value returned by a function must be handled.
- Every public variable should have the same address in every module. An external variable can be defined in one module and can be used in another module. The address of an external variable should be same in every module. Resolving of addresses of symbolic references are handled by the linker.

What is relocation ?

Relocation is the process of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from any designated area of memory.

Consider the statement

MOVER AREG, X

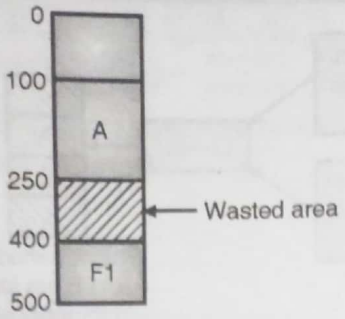
This is an address sensitive instruction. For this instruction to execute correctly, the actual address of X should be put in the instruction.

Assume that a program written in C-language (let us call it A) calls a function F1. The program A and the function F1 must be linked with each other. But when in main storage shall we load A and F1 ? A possible solution would be to load them according to the addresses assigned when they were translated.

Case I

At the time of translation, A has been given storage area from 100 to 250 while F1 occupies area between 400 to 500. If we were to load these programs at their translated addresses, a lot of storage lying between them will be wasted.

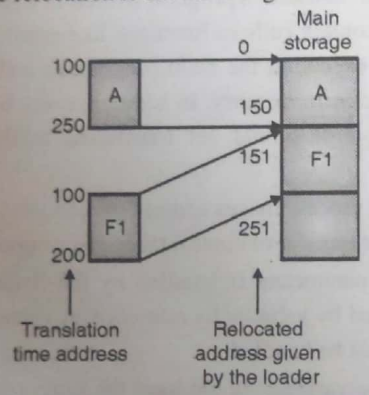
Scanned with CamScanner



(S5.2)**Fig. 1.2.7 : Case I of relocation**

☞ **Case II**

At the time of translation, both A and F_1 may have been translated with the identical start address 100. A goes from 100 to 250 and F_1 goes from 100 to 200. These two modules cannot co-exist at same storage locations. The loader must relocate A and F1 to avoid address conflict or storage waste. A possible relocation is shown in Fig. 1.2.8



(S5.3)**Fig. 1.2.8 : Relocation to avoid address conflict or storage waste**

It may be noted that relocation is more than simply moving a program from one area to another in the storage. It refers to adjustment of address fields and not to movement of a program.

➜ 1.2.4 Macro Processor

Macro allows a sequence of source language code to be defined once and then referred to by name each time it is to be referred. Each time this name occurs in a program, the sequence of codes is substituted at that point.

A macro consists of :

(1) Name of the macro (2) Set of parameters

(3) Body of macro (Code)

Parameters in a macro are optional.

For example, let us consider a program segment given below:

```
ADD AREG, X
ADD BREG, X
==========
ADD AREG, X
ADD BREG, X
==========
ADD AREG, X
ADD BREG, X
==========
```

In the above program, the sequence

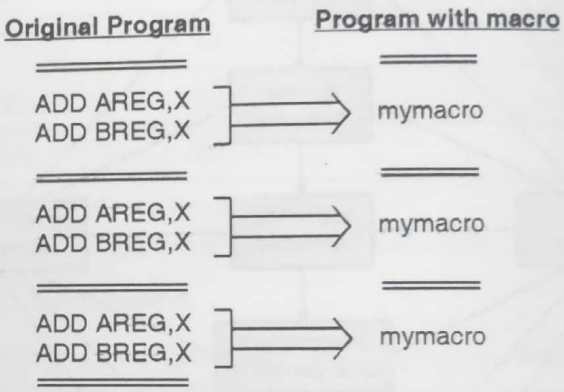
```
ADD AREG, X
ADD BREG, X
```

occurs three times. A macro allows us to attach a name to this sequence and to use this name in its place.

We can attach a name to a sequence by means of a macro definition, which is formed in the following manner :

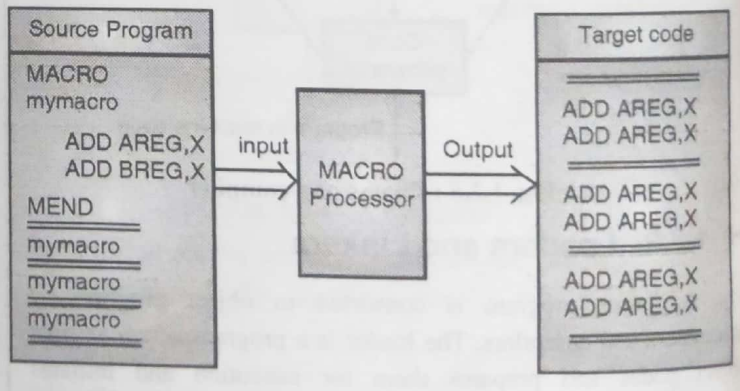
Start of definition	MACRO
macro name	mymacro
macro body	ADD AREG, X ADD BREG, X
End of macro definition	MEND

The example given above can be expressed with macro. It is given in Fig. 1.2.9



(S3.1)**Fig. 1.2.9 : Re-writing a program with macros**

A macro processor takes a source with macro definition and macro calls and replaces each macro call with its body.



(S3.2)**Fig. 1.2.10 : Macro expansion**

➜ 1.2.5 Interpreters

An interpreter is useful during program development. A program execution requires either a compiler or an interpreter. A compiler translates a source program into machine instructions. These machine instructions can be executed by a computer. In case of an interpreter, the interpreter accepts the source program and performs the actions associated with the instructions. Thus it creates a virtual execution environment. The interpreter has the control of the execution of the program during execution.

Scanned with CamScanner

An interpreter reads the source code one instruction or line at a time, converts this line into machine code or some intermediate form and executes it.

The advantage of this is it is simple and you can interrupt it while it is running, change the program and either continue or start again. The disadvantage is that every line has to be translated every time it is executed. Because of this interpreters tend to be slow but it is very useful in program development as the program can be modified on line.

➜ 1.2.6 Operating System

An operating system is an interface between users and the hardware of a computer system. It is a system software which may be viewed as an organised collection of software consisting of procedures for operating a computer and providing an environment for execution of programs.

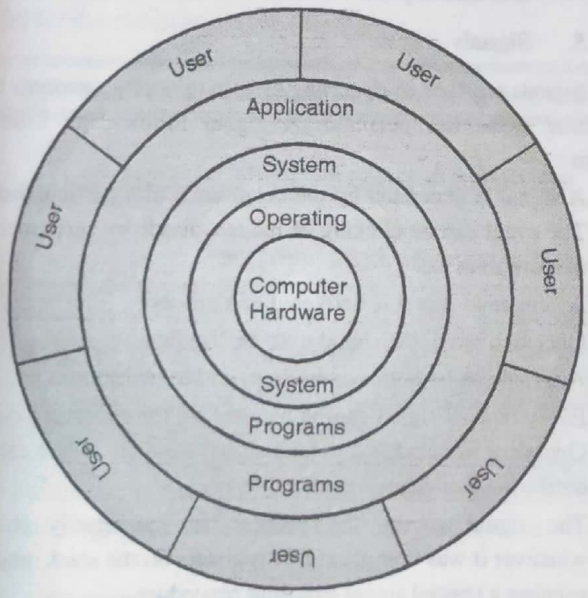
Operating system functions have evolved in response to the following considerations :

1. Efficient utilization of computing resources.
2. New features in computer architecture.
3. New user requirements.

An operating system manages resources of a computer. These resources include :

1. Memory
2. Processor
3. File system.
4. Input/output devices.

It keeps track of the status of each resource and decides who will have a control over computer resources, for how long and when. The positioning of an operating system in overall computer system is shown in the Fig. 1.2.11.



(S7.1)**Fig. 1.2.11 : Components of a computer system**

- Operating system controls computer hardware resources.
- Programs interact with computer hardware with the help of operating system.
- There are two ways in which one can interact with operating system :
  1. By making a system call.
  2. By operating system commands.

1.2.6(a) Operating System Concepts

Some of the important operating system concepts include :

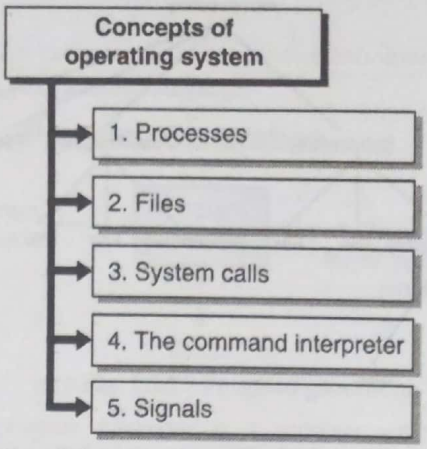


Fig. C1.3 : Concepts of operating system

➜ 1. Processes

A process is program in execution. It consists of the followings :

1. Executable program
2. Program's data
3. Stack and stack pointer
4. Program counter and other CPU registers
5. Details of opened files

- A process can be suspended temporarily and the execution of another process can be taken up. A suspended process can be re-started at a later time.
- Before suspending a process, its details are saved in a table called the process table.
- An operating system supports two system calls to manage processes :
  1. **Create** to create a new process.
  2. **Kill** to delete an existing process.
- A process can create a number of **child processes**.
- Processes can communicate among themselves either using shared memory or by message passing techniques. Two processes running on two different computers can communicate by sending messages over a network.

➜ 2. Files

Files are used for long term storage. Files are used both for input and output. Every operating system provides file management service.

These files hide the peculiarities of the disks and other input/output devices from users. Thus it can also be treated as an abstraction. Every operating system provides system calls for file management. These calls include system calls for :

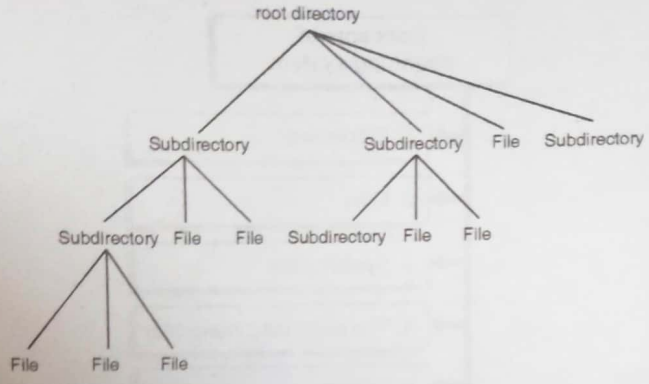
1. File creation
2. File deletion
3. Read and write operations

Files are stored in directory. System calls are provided :

1. To put a file in a directory
2. To remove a file from a directory

Scanned with CamScanner

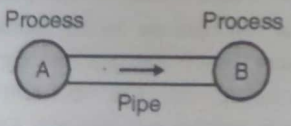
The directory entry may be a file or a directory itself. Directory structure is a hierarchical structure (Shown in Fig. 1.2.12)



(S7.7) **Fig. 1.2.12 : Tree-structured directory**

- Every file can be specified by giving its path from the root.
- At every instant, each process has a current working directory.
- Normally, files in a system are protected so that the privacy of each person's files can be maintained.
- Many operating systems represent each input/output device as a special file. This makes input/output devices look like a file. There are two kinds of special files :
  1. Block special files.
  2. Character special files.

  Block special files can model a device containing a set of randomly accessible blocks. Character special files can model a device consisting of character streams like monitor, keyword etc.
- Many operating systems support a special file known as pipe. Pipe is a Pseudo file that can be used to connector two processes. Output of one process can be sent as input to another process through a pipe. It is shown in Fig. 1.2.13.



(S7.8) **Fig. 1.2.13 : Two processes connected by a pipe**

➜ 3. System calls

System calls provide an interface to the services made available by an operating system. User program interact with operating system through system calls.

- These calls are normally available as library functions in a high level language such as C.
- In an assembly language, necessary parameters are stored in specified CPU-registers and then it issues a software interrupt instruction. System calls in DOS are availed by using "INT 2IH".
- Execution of the following instruction to read n bytes from a file :

```
Count = read (filepointer, buffer, n);
```

Will internally make a system call to read n bytes from the specified file and the data will be stored in the array 'buffer'.

- The number and type of system calls varies from operating system to operating system.

In general, system calls are available for the following operations :

1. Process management
2. Memory management
3. File operations
4. Input/output operations

➜ 4. Command interpreter

There are several ways for users to interface with the operating system. One of the approaches of user interaction with operating system is through commands.

Command interpreter provides command-line interface. It allows user to enter a command on command line prompt. The command interpreter accepts and executes a command entered by an user.

- For example, shell is a command interpreter under UNIX. A $ at the beginning of the command line tells the user that the shell is waiting to accept a command. User can display the contents of the file file1.txt using the following command:

  $ cat file1.txt
- The commands to be executed are implemented in two ways :
  1. Command interpreter itself contains the necessary code to be executed.
  2. Command is implemented through a system file. To execute a command the necessary system file is loaded into memory and executed.

➜ 5. Signals

Signals are used in operating system to notify a process that a particular event has occurred. A signal follows the following pattern :

1. A signal is generated by the occurrence of a particular event. The event can be clicking of mouse, divide by zero operation in a program etc.
2. A generated signal is delivered to a process.
3. Once delivered, the signal must be handled.

- A signal can be both synchronous and asynchronous.
- Every type of signal can be handled by the operating system. Operating system has a default signal handler. A user can also define his own signal handler.
- The signal causes the process to temporarily suspend whatever it was doing, save its registers on the stack, and start running a special signal handling procedure.
- Signals are the software analog of hardware interrupts. Many illegal conditions detected by hardware during program execution are also converted into signals to the guilty process. Signals are also used for inter-process communication.

➜ 1.2.7 Preprocessor

A pre-processor is a program that processes a source file to produce output that is used as input by the translator. The output of

Scanned with CamScanner

a pre-processor is often used by the compiler. Some preprocessors are only capable of performing relatively simple textual substitutions and macro expansions, while others have the power of a full fledged programming language. Some common preprocessors are :

1. Lexical Preprocessors
2. Syntactic preprocessors
3. General purpose preprocessors.

Lexical preprocessors typically perform macro substitution, textual inclusion of other files, and conditional compilation or inclusion. C-preprocessor is an example of lexical pre-processor.

Syntactic preprocessors were introduced with LISP family of languages. They are used for following purposes :

1. Customizing syntax
2. Extending a language
3. Specializing a language.

A pre-processor may by promoted as being general purpose, meaning that it is not aimed at specific usage or programming language :

Example 1.2.1

Compare system program and application program.

Solution :

System software (e.g. compilers, loaders, linkers, macro processors, operating systems) are written to make computers better suited to the needs of computer users. These are helpful and developing an application program. An application runs on top of the operating system and performs a number of tasks for the user application. Software is written to solve problems from application domain. Without application software, a computer will not be useful for the end-user.

Syllabus Topic : Language Processing Activities

1.3 Language Processing Activities

➔ **(Dec. 2013, May 2014, Aug. 2015)**

Q. What are language processing activities ? Give details of language processing activities

SPPU - Dec. 2013, May 2014, 4 Marks

Q. Explain the Language processing activities.

SPPU - Aug. 2015(In Sem), 6 Marks

Language processing activities can be divided into two groups :

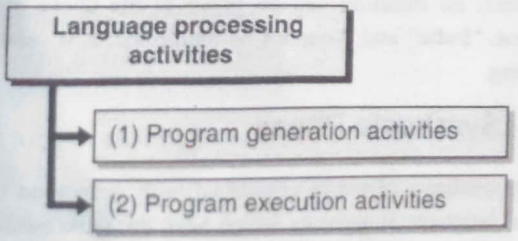
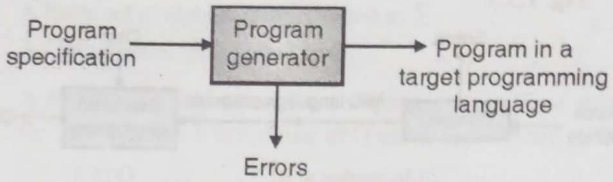


Fig. C1.4 : Language processing activities

Program generation activity can be automated. Here the source language is a specification language of an application domain and the target language is a program in a programming language.

Program execution activities involve execution of a program written in a programming language.

➔ 1.3.1 Program Generation



(S1.6)**Fig. 1.3.1 : Program generation**

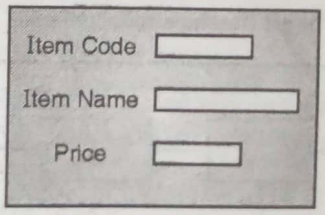
The program generator is a software which accept the specification of a program to be generated, and generates a program in target programming language.

- It is easy to write the specification of a program rather than writing the entire program in a programming language.
- A generated program is more reliable.
- The program generator bridges the gap between application domain and programming language domain.
- It is more economical to develop a program generator than to develop a problem oriented language. Program generators are restricted to specific application domain. It is not possible to develop a generalized program generator.
- The execution gap between the programming language domain and the execution domain is bridged by the compiler or interpreter.

For example, the input form for data entry can be automated with the help of form filling program. User has to give the format of input form with the following details :

(1) Headings.
(2) Type of each field.
(3) Location of entry field on the screen.

The form filling program accepts these details and generates a program in a programming language (say COBOL) for data entry.



(S1.7)**Fig. 1.3.2 : Screen displayed by a screen handling program**

The specification of fields in Fig. 1.3.2 could be as follows :

Item code : Integer : start (line 3, position 25) end (line 3, position 32) ;

Item name : String : start (line 4, position 25) end (line 3, position 35) ;

Price : Float : Start (line 5, position 25) end (line 5, position 35);

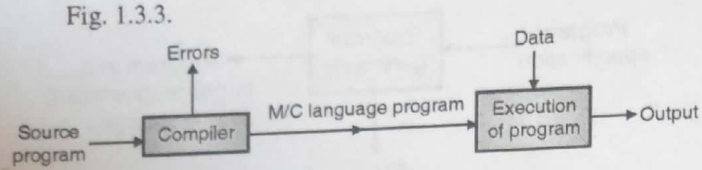
Errors in giving specifications are located by the program generator.

Scanned with CamScanner

1.3.2 Program Translation / Execution

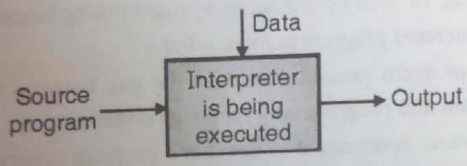
A program translation activity involves translation of a program from source code into machine code.

- Compilers are used for program translation.
- The translated program may be saved in a file. The saved program may be executed repeatedly.
- The program execution using a compiler is shown in Fig. 1.3.3.



(S1.8) **Fig. 1.3.3 : Program execution using a compiler**

A program can also be executed by an interpreter. An interpreter reads the source code one instruction at a time converts this instruction into machine code or some intermediate form and executes it. The program execution using an interpreter is shown in Fig. 1.3.4.



(S1.9) **Fig. 1.3.4 : Program execution using an interpreter**

Syllabus Topic : Fundamentals of Language Processing

1.4 Fundamentals of Language Processing

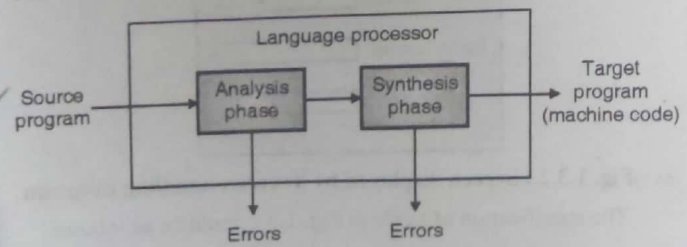
➔ **(Dec. 2015)**

Q. Explain the different phases of language processing. **SPPU - Dec. 2015, 6 Marks**

A language processing involves :

(1) Analysis of source program.

(2) Synthesis of target (machine code) program.



(S1.10) **Fig. 1.4.1 : Phases of language processing**

1.4.1 Analysis Phase

➔ **(May 2015, May 2016)**

Q. Explain Analysis phase of compiler. **SPPU - May 2015, May 2016, 7 Marks**

A source program consists of several components. These components include :

(1) **Reserve words** like int, float, void, char, while etc.

(2) **Statements** like assignment statement, while loop etc.

(3) **Constants** like integer constants (153, 94), floating point constants (15.64, 93.845), character constants ('A', '9') etc.

These components are written using predefined rules. They have to meet language specifications.

Analysis of a source statement consists of the followings :

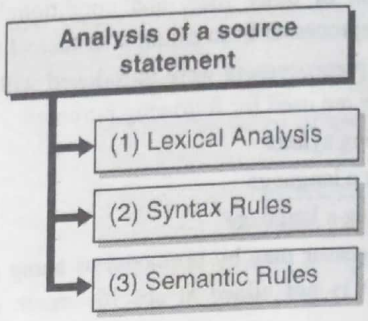


Fig. C1.5: Analysis of a source statement

➔ **(1) Lexical Analysis**

Lexical analysis rules govern the formation of lexical units (symbols) in the source program.

☞ **Example**

(1) Rules for identifier name.

(2) Rules for integer constant.

(3) Rules for string constants.

An identifier name must start with an alphabet or underscore followed by a list of alphanumeric characters. x1 is a valid identifier, but 1x is not a valid identifier.

➔ **(2) Syntax Rules**

Syntax rules govern the formation of valid statement in the source language.

☞ **Example**

(1) x = y + 3 ; is a valid statement in C-Language.

(2) x = y + *3 ; is not a valid statement in C-language.

➔ **(3) Semantic Rules**

Semantic rules associate meaning with valid statement of the language.

☞ **Example**

x = "India" *3 ; Although the statement is syntactically correct, no meaning can be given to the above statement. Since, "India" and 3 cannot be multiplied, it is semantically wrong.

1.4.2 Synthesis Phase

The synthesis phase is concerned with generation of target (machine) language statements which have the same meaning as a source statement. A synthesis phase is close to machine domain.

Scanned with CamScanner

1.4.3 Forward References and Passes

➔ **(May 2013, Oct. 2016, Dec. 2016)**

Q. Define the term forward reference and explain where it is used with example. **SPPU - May 2013, 2 Marks**

Q. Explain the terms terminals, non-terminals, starting symbol and production rule set with example. **SPPU - Oct. 2016(In Sem), 6 Marks**

Q. Explain with example use of Terminals and non-terminals in representing language grammer. **SPPU - Dec. 2016, 6 Marks**

During synthesis phase, every variable should be assigned memory address. A machine instruction uses memory address to reference a variable. Therefore, the generation of machine code must be delayed until addresses of every variable are fixed.

A forward reference of a program entity is a reference to the entity which precedes its declaration.

For example, in the code given below

```
x = y + 5 ;
int x, y ;
```

The statement "x = y + 5 ;" makes forward reference to variables x and y as they are declared later in the program.

The compiler will not be able to generate the machine code for the statement.

"x = y + 5 ;" until it has seen declaration of the two variables x and y.

- Passes are required to solve the above problem.
- Compiler will scan the source code twice (two passes).

☞ **Pass I**

Perform analysis of the source program store necessary information. For the above example, it will look for variable declaration before going to pass II.

☞ **Pass II**

Perform synthesis of target program. It is same as generation of machine code.

1.5 Fundamentals of Language Specification

A source program analysis is based on source language specification. During analysis phase, a translator (compiler) must ensure that :

(1) Program meets lexical rules.
(2) Program meets syntactic rules.
(3) Program meets semantic rules.

To write an analysis phase of a compiler, one should have the knowledge of the following :

(1) Finite automata
(2) Regular expression
(3) Context free grammar

1.5.1 Deterministic Finite Automata (DFA)

The word "deterministic" refers to the fact that the transition is deterministic. There is only one state to which an automaton can transit from the current state on each input. The word "finite" implies that number of states is finite. A finite automata consists of five parts :

(1) A finite set of states, represented as Q.
(2) A finite set of alphabet, represented as Σ.
(3) An initial state, represented as q_0.
(4) A set of accepting states. An accepting state or final state is for 'yes' answer. It is a subset of Q and is represented as F.

$F \subseteq Q$ [F is subset of Q]

(5) A next state function or a transition function. Next state depends on the current state and the current input. It is a function from $Q \times \Sigma$ to Q. It is represented as δ.

1.5.1(a) Definition of a DFA ➔ (May 2013)

Q. Define the term DFA and explain where it is used with example. **SPPU - May 2013, 2 Marks**

A deterministic finite automata is a quintuple.

$$M = (Q, \Sigma, \delta, q_0, F), \text{ where}$$

Q is a set of states.
Σ is a set of alphabet.
$q_0 \in Q$ is the initial state,
$F \subseteq Q$ is the set of final states, and δ, the transition function, is a function from $Q \times \Sigma$ to Q.

1.5.1(b) Representation of a DFA

Let the machine M be a deterministic finite automata.

$$M = \{Q, \Sigma, \delta, q_0, F\}, \text{ where}$$

$$Q = \{q_0, q_1\}$$

$$\Sigma = (0, 1)$$

q_0 is the starting state

$$F = \{q_1\}.$$

and δ is the transition function as given below :

$\delta(q_0,0) \Rightarrow q_0$.
$\delta(q_0,1) \Rightarrow q_1$
$\delta(q_1,0) \Rightarrow q_1$
$\delta(q_1,1) \Rightarrow q_0$.

The above representation of transition function is not very readable. Conventionally, there are two representations for transition function :

(1) State transition table.
(2) State transition diagram.

(1) State transition table

The Fig. 1.5.1 shows the state transition table of the transition function discussed in this section.

Scanned with CamScanner

	0	1
$\rightarrow q_0$	q_0	q_1
q_1^*	q_1	q_0

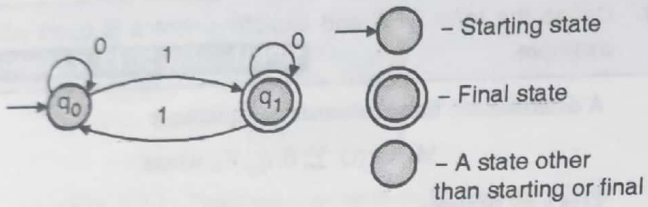
Input alphabets.
$\rightarrow$ before q_0 indicates it is the starting state
* indicates a final state.
states

Fig. 1.5.1 : State transition table

- In state transition table, there is a row for every state $q_i \in Q$.
- In state transition table, there is a column for every alphabet $A_i \in \Sigma$.
- Starting state is marked as '$\rightarrow$'.
- Final state is marked as '*'.

(2) State transition diagram

The Fig. 1.5.2 shows the state transition diagram of the transition function discussed in this section.



(S1.11)**Fig. 1.5.2 : State transition diagram**

Language of the above DFA

- It is easy to see that machine remain in same state on input 0.
- The machine transitions from q_0 to q_1 and q_1 to q_0, when the input is 1.
- Machine will be in state q_1 (final state), after reading odd number of 1's.

Thus we can conclude that the DFA accepts a string if and only if the number of 1's is odd.

The language L accepted by M, the given DFA, is the set of all strings having odd number of 1's.

Or,

$L(M) = \{\omega \in \{0, 1\}^* \mid \omega$ contains odd number of 1's$\}$

Simulating the functioning of the above DFA for a given input

If M is given an input 0010110, its initial configuration is $(q_0, 0010110)$.

$(q_0, 0010110) \overset{\delta}{\Rightarrow} (q_0, 010110)$ [input 0 in state q_0]

$\overset{\delta}{\Rightarrow} (q_0, 10110)$ [input 0 in state q_0]

$\overset{\delta}{\Rightarrow} (q_1, 0110)$ [input 1 in state q_0]

$\overset{\delta}{\Rightarrow} (q_1, 110)$ [input 0 in state q_1]

$\overset{\delta}{\Rightarrow} (q_0, 10)$ [input 1 in state q_1]

$\overset{\delta}{\Rightarrow} (q_1, 0)$ [input 1 in state q_0]

$\overset{\delta}{\Rightarrow} (q_1, \varepsilon)$ [input 0 in state q_1]

Therefore,

$(q_0, 0010110) \overset{\delta^*}{\Rightarrow} (q_1, \varepsilon)$ and so the input 0010110 is accepted as q_1 is a final state.

1.5.1(c) Designing a DFA

Designing a DFA is like writing a program. In a program, the current state of a program is given by :

(1) Values stored in variables.

(2) Current line of execution.

In case of a DFA, variables are not used. Everything is remembered through explicit states. Transition function of a DFA is similar to algorithm. While designing a DFA, one has to concentrate on four issues.

These are :

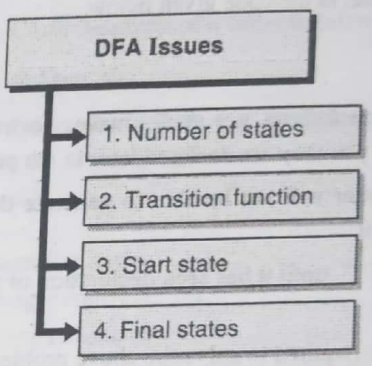


Fig. C1.6 : Issues to be considered while designing DFA

1. Number of states

Number of states depends on "what must be remembered as input symbols are read by a DFA."

For example,

(1) DFA to check whether a binary numbers has even number of 1's :

DFA has to remember whether the number of 1's seen so for is even or odd. It does not have to count number of 1's. Let us look at a binary number given below :

Binary number →	0	1	1	0	1	1	0	0	1	1
Number of 1's →	even	odd	even	even	odd	even	even	even	odd	even

- Machine will have two states :
  - (i) One corresponding to even number of 1's, so far.
  - (ii) One corresponding to odd number of 1's, so far.
- These two states are necessary and sufficient to conclude whether a binary number contains even number of 1's.

(2) DFA to check whether a string over alphabets (a, b) contains substring abb :

DFA has to search for substring abb in input stream. Input string is not stored in a memory variable. Input is made one symbol at a time. As the substring is of length 3, it may be necessary to remember the preceding two symbols.

- Preceding symbol is a. (a is the first symbol in abb).

Scanned with CamScanner

- Preceding two symbols are ab. (ab constitutes first two symbols of abb).
- DFA has already seen the substring abb in the input string. Once the substring abb is found in input string, this status will not change irrespective of what follows thereafter in the input string.
- Preceding symbols are neither a nor ab, and abb has not come earlier.

The machine will have four states, each standing for one of the four situations.

➜ 2. Transition function

Transition function gives the next-state, depending on :

1. Current state 2. Current input

A transition function is problem specific and it depends on the problem.

☞ For example

(1) Give transition function for a "DFA to check whether a binary number has even number of 1's".

We have already seen that the corresponding DFA will have two states :

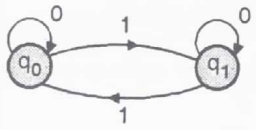
(i) State q_0, indicating even number of 1's seen so far.

(ii) State q_1, indicating odd number of 1's seen so far.

- 0 as next input will have no effect on number of 1's.
- 1 as next input will make the transition from q_0 to q_1 and from q_1 to q_0. Thus, the transition behaviour can be described using the Fig. 1.5.3.

$\delta(q_0,0) \Rightarrow q_0$
$\delta(q_0,1) \Rightarrow q_1$
$\delta(q_1,0) \Rightarrow q_1$
$\delta(q_1,1) \Rightarrow q_0$

	0	1
q_0	q_0	q_1
q_1	q_1	q_0



(a) Tabular form (b) Transition table (c) Transition diagram

(S1.12) **Fig. 1.5.3 : Transition behaviour**

(2) Give transition function for a "DFA to check whether a string over alphabets (a, b) contains a substring abb".

We have already seen that the corresponding DFA will have four states.

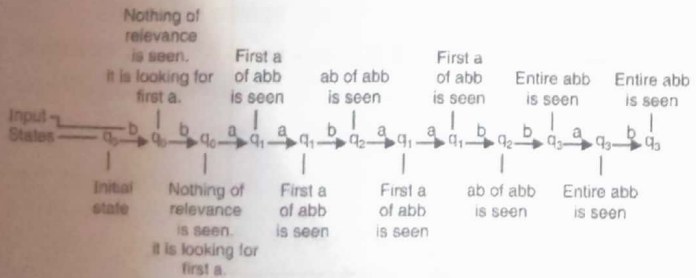
(1) State q_1, preceding symbol is a.

(2) State q_2, preceding two symbols are ab.

(3) State q_3, substring abb has already been seen in input string.

(4) State q_0, situations q_1, q_2, or q_3 are not there.

State of the DFA after every input symbol for a sample input data is shown in Fig. 1.5.4.



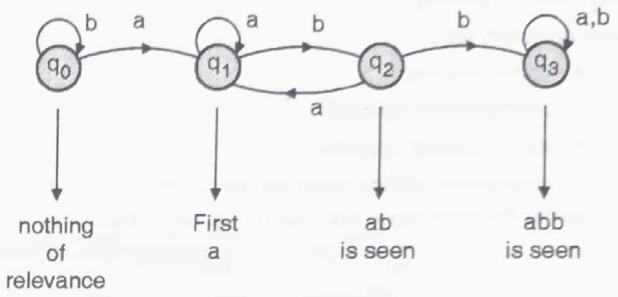
(S1.13) **Fig. 1.5.4 : States of DFA**

Transition behaviour is described using Fig. 1.5.5.

$\delta(q_0,b) \Rightarrow q_0$
$\delta(q_0,a) \Rightarrow q_1$
$\delta(q_1,a) \Rightarrow q_1$
$\delta(q_1,b) \Rightarrow q_2$
$\delta(q_2,a) \Rightarrow q_1$
$\delta(q_2,b) \Rightarrow q_3$
$\delta(q_3,a) \Rightarrow q_3$
$\delta(q_3,b) \Rightarrow q_3$

	a	b
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_2
q_3^*	q_3	q_3

(a) Tabular form (b) Transition table



(c) Transition diagram

(S1.14) **Fig. 1.5.5 : Transition behaviour**

- Machine transits from q_0 to q_1 on input a, as the first symbol a of abb is seen.
- Machine remains in q_1 state on input a, as the previous two symbols aa will still make the first symbol a of abb.
- Machine transits from q_1 to q_2 on input b, as ab of abb is seen.
- Machine transits from q_2 to q_3 on input b, as entire abb is seen.
- Machine transits from q_2 to q_1 on input a, as the previous three symbols aba will still make the first symbol a of aba.

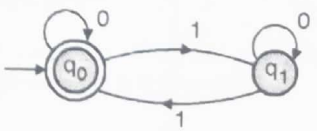
➜ 3. Starting state and final states

In Fig. 1.5.3(c), q_0 is both the initial state and final state as :

(1) Zero number of 1's is even number of 1's.

(2) Machine will have "yes" answer for even number of 1's. q_0 is final state. Machine in Fig. 1.5.3(c) is redrawn in Fig. 1.5.6 with initial and final states marked.

	0	1
$\rightarrow q_0^*$	q_0	q_1
q_1	q_1	q_0



(S1.15) **Fig. 1.5.6 : Final DFA for Fig. 1.5.3**

In Fig. 1.5.5(c), q_0 is initial state and q_3 is the final state.

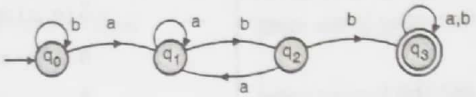
(1) Initially, we have nothing that is relevant to substring abb. Therefore, q_0 is initial state.

(2) Machine goes to state q_3 after seeing the substring abb. Therefore, q_3 is final state.

Machine in Fig. 1.5.5 is redrawn in Fig. 1.5.7 with initial and final states marked.

Scanned with CamScanner

	a	b
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_3
q_3^*	q_3	q_3



(S1.16)**Fig. 1.5.7 : Final DFA for Fig. 1.5.5(c)**

1.5.2 Regular Expression

➔ (May 2013)

Q. Define the term regular expression and explain where it is used with example. **SPPU - May 2013, 2 Marks**

The set of strings accepted by finite automata is known as regular language. This language can also be described in a compact form using a set of operators.

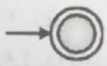
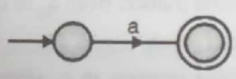
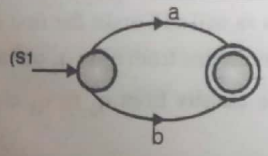
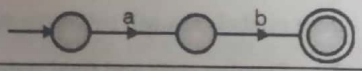
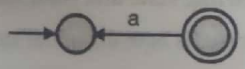
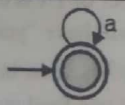
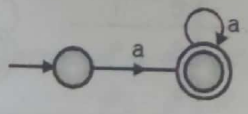
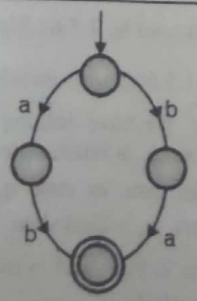
These operators are :

(1) + , union operator

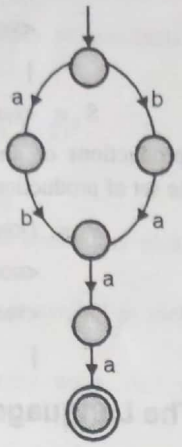
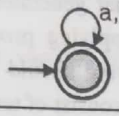
(2) . , concatenation operator

(3) * , star or closure operator.

An expression written using the set of operators (+, · , *) and describing a regular language is known as regular expression. Regular expressions for some basic automata are given in Fig. 1.5.8.

Automata	Language	Regular expression
	{ε}	R.E. = ε
	{a}	R.E. = a
	{a, b}	R.E = a + b
	{ab}	R.E. = a · b or simply ab
	ϕ	R.E. = ϕ
	{ε, a, aa, aaa, ...}	R.E. = a*
	{a, aa, aaa, ...}	R.E. = aa* or a+
	{ab, ba}	R.E. = ab + ba

Scanned with CamScanner

Automata	Language	Regular expression
	{abaa, baaa}	R.E. = (ab + ba) aa
	{ε, a, b, aa, ab, ba, bb, ...}	R.E. = (a + b)*

(S1.17)**Fig. 1.5.8 : Examples on regular expression**

If R_1 and R_2 are regular expressions then :

$R_1 + R_2$ is also regular,

$R_1 \cdot R_2$ is also regular,

R_1^* is also regular,

R_2^* is also regular,

R_1^+ is also regular.

[R_1^+ Stands for one or more occurrences of R_1]

- 0* stands for a language in which a word contains zero or more 0's.
- (0 + 1)* stands for a language in which a word ω contains any combination of 0's and 1's and $|\omega| \geq 0$.

Example 1.5.1

Write regular expression for the following languages.

(a) The set {1010}

(b) The set {10, 1010}

(c) The set {ε, 10, 01}

(d) The set {ε, 0, 00, 000, ...}

(e) The set {0, 00, 000, ...}

(f) The set of strings over alphabet {0, 1} starting with 0.

(g) The set of strings over alphabet {0, 1} ending in 1.

(h) The set of strings over alphabet (a, b} starting with a and ending in b.

(i) The set of strings recognized by $(a + b)^3$

Solution :

(a) Regular expression,

R.E. = 1010 stands for the set {1010}

(b) Regular expression,

R.E. = 10 + 1010

represent the set {10, 1010}

(c) The set {ε, 10, 01} is represented by the regular expression,

R.E. = ε + 10 + 01

(d) The set {ε, 0, 00, 000, ...} is represented by the regular expression,

R.E. = 0*

(e) The set {0, 00, 000, ...} is represented by the regular expression,

R.E. = 0^+

R.E. = 00*

00* represents 0{ε, 0, 00, 000, ...} = {0, 00, 000, ...}

(f) The set of string starting with 0 is given by 0 followed by any combination of 0, 1.

R.E. = 0(0 + 1)*

(g) The regular expression for a set over alphabet {0, 1} ending in 1 is given by :

R.E. = (0 + 1)*1

(h) The regular expression for a set of strings over alphabet {a, b} starting with a and ending in b is given by,

R.E. = a(a + b)*b

(i) $(a + b)^3$ stands for (a + b) (a + b) (a + b) = aaa + aab + aba + abb + baa + bab + bba + bbb.

1.5.3 Context Free Grammar → (May 2013)

Q. What do you understand by Grammar? Explain the use of Terminal and Non-Terminal in representing grammar.

SPPU - May 2013, 4 Marks

Scanned with CamScanner

A context free grammar G is a quadruple (V, T, P, S),

where,

V is a set of variables.

T is a set of terminals.

P is a set of productions.

S is a special variable called the start symbol $S \in V$.

A production is of the form

$V_i \rightarrow \alpha_i$ where $V_i \in V$ and α_i is a string of terminals and variables.

1.5.3(a) Notations

(1) Terminals are denoted by lower case letters a, b, c … or digits 0, 1, 2 …. etc.

(2) Non-terminals (variables) are denoted by capital letters A, B, …, V, W, X …

(3) A string of terminals or a word $w \in L$ is represented using u, v, w, x, y, z.

(4) A sentential form is a string of terminals and variables and it is denoted by α, β, γ etc.

☞ CFG explained through an example

Let us consider English sentences of the form :

(1) Mohan eats. (2) Soham plays.

(3) Ram reads.

The first word of in the above sentences is a noun and the second word is a verb. A sentence of the above form can be written as

<sentence> → <noun> <verb>

Here, noun can be replaced with Mohan, Soham or Ram and <verb> can be replaced with reads, plays or eats. We can write :

<noun> → Mohan

<noun> → Soham

<noun> → Ram

<verb> → eats

<verb> → plays

<verb> → reads

In the above example :

(1) <sentence>, <noun> and <verb> are variables or non-terminals.

(2) Mohan, Soham, Ram, eats, plays and reads are terminals.

(3) The variable <sentence> is the start symbol as a sentence will be formed using the start symbol <sentence>.

Formally, the grammar can be written as :

G = (V, T, P, S), where

V = {<sentence>, <noun>, <verb>}

T = {Mohan, Soham, Ram, eats, plays, reads}

P = {<sentence> → <noun> <verb>,
<noun> → Mohan,
<noun> → Soham,
<noun> → Ram,
<verb> → eats,
<verb> → plays,
<verb> → reads
}

S = <sentence>

Several productions of <noun> and <verb> can be merged together and the set of productions can be re-written as :

P = {<sentence> → <noun> <verb>,
<noun> → Mohan | Soham | Ram,
<verb> → eats | plays | reads
}

1.5.3(b) The Language of a Grammar

Every grammar generates a language. A word of a language is generated by applying productions a finite number of times. Derivation of a string should start from the start symbol and the final string should consist of terminals.

If G is a grammar with start symbol S and set of terminals T, then the language of G is the set

$$L(G) = \left\{ w \mid w \in T^* \text{ and } S \overset{*}{\underset{G}{\Rightarrow}} w \right\}$$

If the production rule is applied once, then we write $\alpha \underset{G}{\Rightarrow} \beta$.

When it is applied a number of times then we write $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$

Derivations are represented either in the

(1) Sentential form OR

(2) Parse tree form.

1.5.3(c) Sentential Form

Let us consider a grammar given below :

S → A1B (Production 1.1)

A → 0A | ε (Production 1.2)

B → 0B | 1B | ε (Production 1.3)

where, G is given by (V, T, P, S)

with, V = {S, A, B}

T = {0, 1}

P = {Productions 1.1, 1.2 and 1.3}

S = Start symbol

Let us try to generate the string 00101 from the given grammar.

S → A1B [Starting production]

→ 0A1B [Using the production A → 0A]

→ 00A1B [Using the production A → 0A]

→ 001B [Using the production A → ε]

→ 0010B [Using the production B → 0B]

→ 00101B [Using the production B → 1B]

→ 00101 [Using the production B → ε]

Scanned with CamScanner

Thus the string 00101 ∈ L(G).

In sentential form, derivation starts from the start symbol through a finite application of productions.

A string α derived so far consists of terminals and non-terminals.

$$S \underset{G}{\overset{*}{\Rightarrow}} \alpha \mid \alpha \in (V \cup T)^*$$

- A final string consists of terminals.
- In left sentential form, leftmost symbol is picked up for expansion.
- In right sentential form, rightmost symbol is picked up for expansion.
- A string can be derived in many ways. But we restrict ourselves to :
  (1) Leftmost derivation.
  (2) Rightmost derivation.

In leftmost derivation the leftmost variable of α (sentential form) is picked for expansion.

In rightmost derivation the rightmost variable of α (sentential form) is picked for expansion.

Example 1.5.2

For the grammar given below

S → A1B

A → 0A | ε

B → 0B | 1B | ε

Give leftmost and rightmost derivation of the string 1001.

Solution :

(i) Leftmost derivation of 1001 (Leftmost variable is picked up for expansion.)

S → A1B [Derivation starts from the start symbol]
→ 1B [Using the production A → ε]
→ 10B [Using the production B → 0B]
→ 100B [Using the production B → 0B]
→ 1001B [Using the production B → 1B]
→ 1001 [Using the production B → ε]

(ii) Rightmost derivation of 1001 (Rightmost variable is picked up for expansion)

S → A1B [Derivation starts from the start symbol]
→ A10B [Using the production B → 0B]
→ A100B [Using the production B → 0B]
→ A1001B [Using the production B → 1B]
→ A1001 [Using the production B → ε]
→ 1001 [Using the production A → ε]

1.5.3(d) Parse Tree

A set of derivations applied to generate a word can be represented using a tree. Such a tree is known as a parse tree. A parse tree representation gives us a better understanding of :

(1) Recursion
(2) Grouping of symbols

A parse tree is constructed with the following condition :

(1) Root of the tree is represented by start symbol.
(2) Each interior mode is represented by a variable belonging to V.
(3) Each leaf mode is represented by a terminal or ε.

A string generated by a parse tree is seen from left to right.

Example 1.5.3

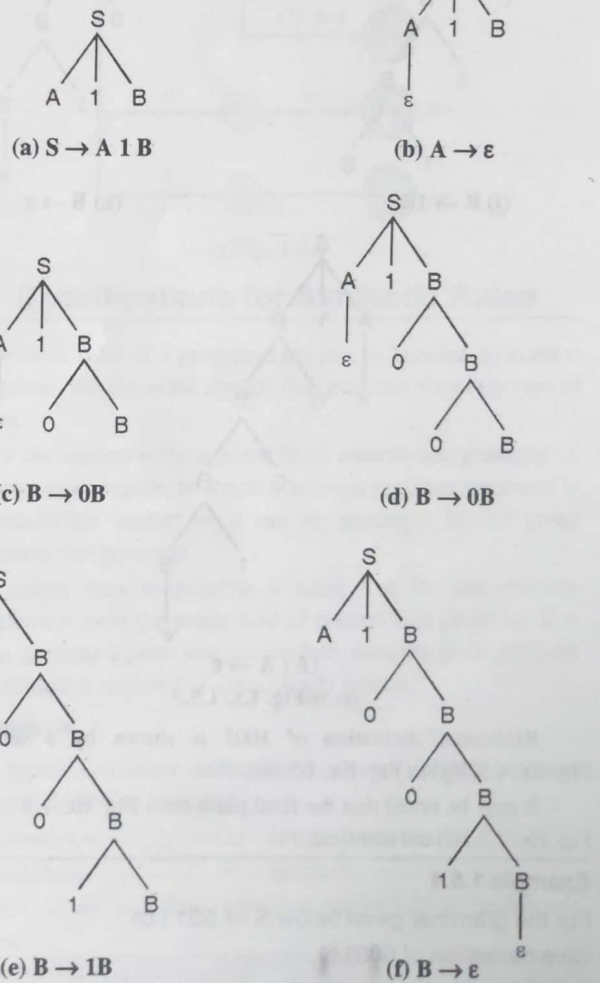
For the grammar given below

S → A1B ; A → 0A | ε ; B → 0B | 1B | ε

Give parse tree for leftmost and rightmost derivation of the string 1001.

Solution :

(i) Parse tree for leftmost derivation of 1001.

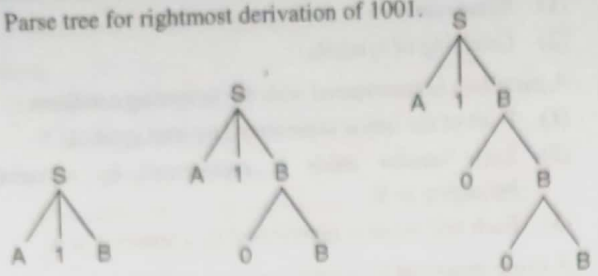


(S1.18) **Fig. Ex. 1.5.3**

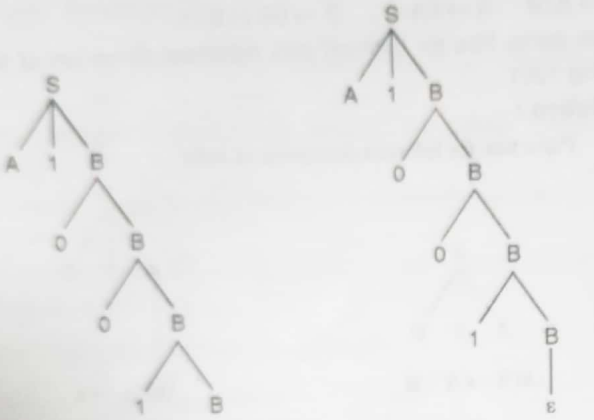
Leftmost derivation of 1001 is shown by series of Figs. 1.5.3(a) to 1.5.3(f).

Scanned with CamScanner

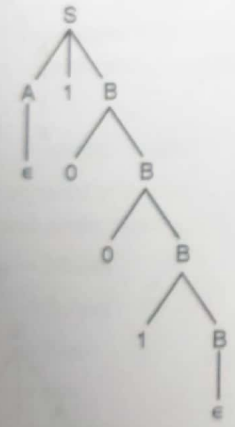
(ii) Parse tree for rightmost derivation of 1001.



(g) S → A 1 B (h) B → 0B (i) B → 0B



(j) B → 1B (k) B → ε



(l) : A → ε

(S1.18) **Fig. Ex. 1.5.3**

Rightmost derivation of 1001 is shown by a series of Fig. Ex. 1.5.3(g) to Fig. Ex. 1.5.3(*l*).

It may be noted that the final parse trees Fig. Ex. 1.5.3(f) and Fig. Ex. 1.5.3(*l*) are identical.

Example 1.5.4

For the grammar given below S → 0S1 | 01

Give derivation of 000111.

Solution :

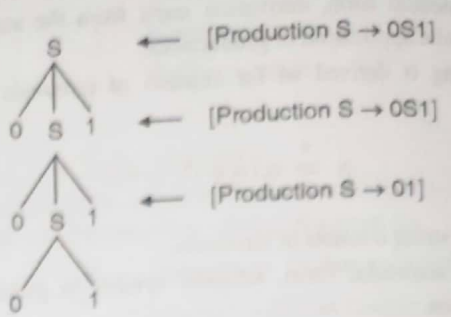
(1) Derivation in sentential form

S → 0S1 [Using production S → 0S1]

S → 00S11 [Using production S → 0S1]

S → 000111 [Using production S → 01]

(2) Derivation using parse tree.



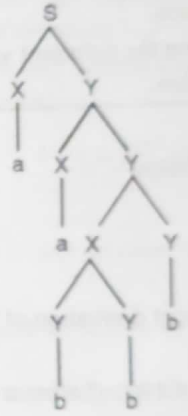
(S1.19) **Fig. Ex. 1.5.4 : Derivation of 000111**

Example 1.5.5

Find whether the string aabbb is in L = L(G), where G is given by S → XY, X → YYla, Y →XY | b

Solution :

Parse tree for the string aabbb is shown in Fig. Ex. 1.5.5.



(S1.20) **Fig. Ex. 1.5.5 : Derivation of aabbb**

Hence, aabbb ∈ L(G).

Example 1.5.6

For the grammar given below

E → E + T | T

T → T * F | F

F → (E) | a | b

Give the derivation of (a + b) * a + b.

Solution :

For the grammar given in Example 1.5.6,

Set of variables V = {E, T, F}

Set of terminals Σ = {+, *, (,), a, b}

$$\text{Set of productions P} = \left\{ \begin{array}{l} E \rightarrow E + T \mid T, \\ T \rightarrow T * F \mid F, \\ F \rightarrow (E) \mid a \mid b \end{array} \right\}$$

Start symbol = E

(i) Derivation in sentential form

E → E + T [Using production E → E + T]

→ T + T [Using production E → T]

→ T * F + T [Using production T → T * F]

Scanned with CamScanner

→ F * F + T [Using production T → F]

→ (E) * F + T [Using production F → (E)]

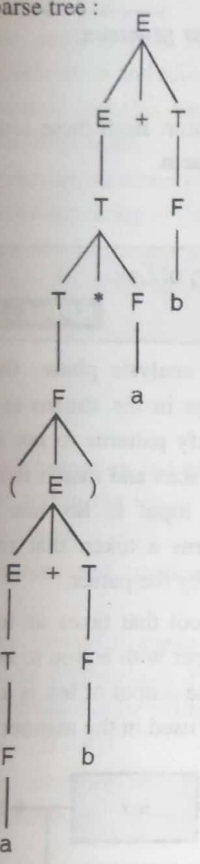
→ (E + T) * F + T [Using production E → E + T]

→ (T + T) * F + T [Using production E → T]

→ (F + F) * F + F [Using production T → F]

→ (a + b) * a + b [Using production F → a | b]

(ii) Derivation using parse tree :



(S1.21)**Fig. Ex. 1.5.6 : Parse tree for (a + b) * a + b**

1.6 Specifications for Lexical Rules

Each lexical unit in the source language can be expressed using regular expression. For every regular expression there is an equivalent DFA (deterministic finite automata). DFA can be viewed as an algorithm for recognizing a lexical unit in the source language.

Example

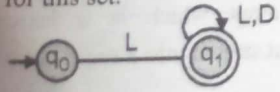
Identifiers is some programming languages is the set of strings of letters and digits beginning with a letter. Here is a regular definition for this set.

L → A + B + ... + Z + a + b + ... + z

D → 0 + 1 + ... + 9

Identifier → L (L + D)*

Here is a DFA for this set.



(S1.22)**Fig. 1.6.1**

Where L is a set of alphabet and D is a set of digits.

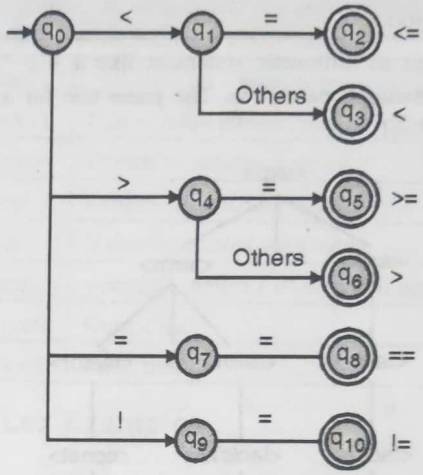
It may be noted that a C-Program can be generated automatically that can simulate a DFA. We can always consider a DFA as an algorithm. Similarly, a DFA can be generated automatically from a regular expression.

Example

Every programming language has a set of relational operators. Here is a regular definition for relational operators of C-Language.

Relop → < + > + == + < = + > = + ! =

Here is a DFA for this set.



(S1.23)**Fig. 1.6.2**

1.7 Specifications for Syntactic Rules

Syntactic rules of a programming can be handled by context free grammar. We can write context free grammar for every type of statement.

- We can always write a parser for a context free grammar. A parser is a program to check whether a program statement is syntactically correct or it can be generated by the given context free grammar.
- A parser tries to generate a parse tree for the program statement from the production of context free grammar. If it can generate a parse tree successfully then the given program statement is declared as syntactically correct.

Example

A typical arithmetic statement in a programming language contains the following (assumption) :

(1) Operators +, –, *, / (2) Integer constants

(3) Identifiers. (4) Brackets.

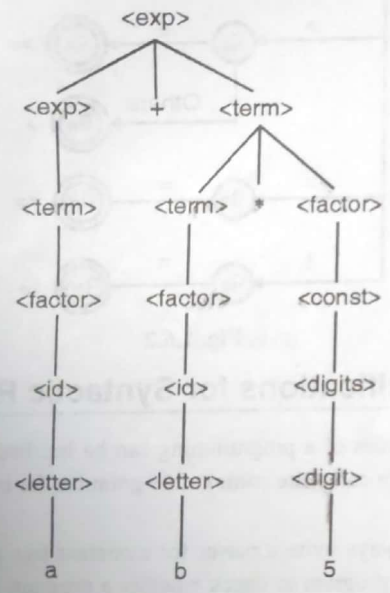
Here is context free grammar for a set of arithmetic statements.

<exp> → <exp> + <term>
| <exp> – <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <factor>
<factor> → (<exp>) | <id> | <const>

Scanned with CamScanner

<id>→<letter><alphanum>
<alphanum>→<letter><alphanum>
|<digit><alphanum>
|<letter>
|<digit>
<letter> → A | B | ... | Z | a | b | ... |z
<digits> → <digit> <digits>
| <digit>
<digit> → 0 | 1 | ... |9
<const> → + <digits>
| – <digits>
| <digits>

The syntax of arithmetic statement like a + b * 5 can be checked by drawing a parse tree. The parse tree for a + b *5 is shown in Fig. 1.7.1.



(S1.24)**Fig. 1.7.1 : Parse tree for a + b * 5**

1.8 Language Processor Development Tools

➔ **(May 2013, May 2014, May 2015, May 2016)**

Q. What are language processor development tools?
SPPU - May 2013, 4 Marks

Q. Explain different development tools used to develop language processor.
SPPU - May 2014, May 2015, May 2016, 6 Marks

Writing a compiler is not a simple project and anything that makes the task simpler is worth exploring. Many aspects of compiler design can be automated. There are several software tools that help in the production of a compiler. Two best known software tools for compiler construction are :

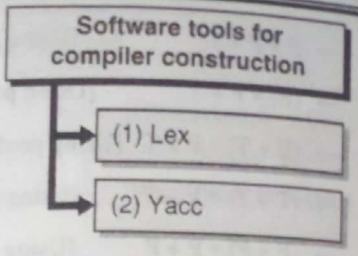


Fig. C1.7 : Software tools

➔ **(1) lex**

It is a lexical analyzer generator.

➔ **(2) yaac**

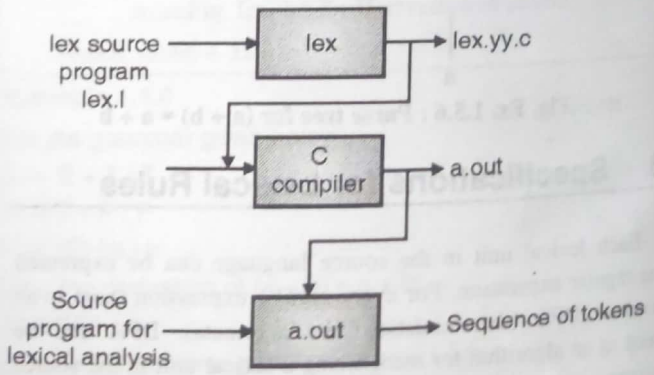
It is a parser generator. Both these tools are available under UNIX operating system.

1.8.1 Lex ➔ (May 2013)

Q. Explain the working of Lex.
SPPU - May 2013, 2 Marks

During the lexical analysis phase, the compiler reads the input and converts strings in the stream to tokens. With regular expressions we can specify patterns to **lex** so that it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to **lex** has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser.

Lex is a software tool that takes as input a specification of regular expressions together with action to be taken on recognising each of these patterns. The output of **lex** is a C-Program for lexical analysis. Lex is generally used in the manner shown in Fig. 1.8.1.



(S1.25)**Fig. 1.8.1 : Creating a lexical analyzer with lex**

- First, a specification of lexical analyzer as prepared by creating a program lex·l in the Lex language.
- Then, lex·l is run through the lex compiler to produce a C-Program lex·yy·c.
- lex.·yy·c is a C-Program containing recognizer for regular expressions together with user supplied code.
- Finally, lex·yy·c is run through the C-Compiler to produce an object program a·out which is a lexical analyzer that transforms an input stream into a sequence of tokens.

Scanned with CamScanner

Lex specifications

A Lex program consists of three parts :

```
declarations
%%
translation rules
%%
user functions
```

- Any of these three sections may be empty but the %% separator between declaration and rules cannot be omitted.
- The declaration section includes declaration of variables, constants and regular definitions.
- The regular definitions are statements used as components of the regular expressions appearing in the translation rules.
- The translation rules of a lex program which is a key part of the lex input are statements of the form :

  P1 [action]
  P2 [action]
  ⋮
  Pn [action]

Where Pi is a regular expression and each action is a program fragment, describing what action the lexical analyzer should taken when pattern Pi matches a token.

- The third section contains user functions which are needed by 'actions.'
- Lex supports a very powerful range of operators for the construction of regular expressions. For example a regular expression for an identifier can be written as :

  [A – Za – z] [A – Za – Z0 – 9]*

which represents an arbitrary string of letters and digits beginning with a letter suitable for matching a variable in many programming languages.

Here is a list of lex operators with examples :

Operator notation	Example	Meaning
* (asterisk)	a*	Set of all strings of zero or more a's, i.e. {E, a, a, a, ...}
| (or)	a | b	Either a or b
+	a+	Set of all strings of one or more a's, i.e. {a, a, a, ...}
?	a?	Zero or one instance of a.
[ ]	[abc]	Either a or b or c. An alphabetical class such as [a..z] denotes the regular expression a|b|...|z.

Here are few examples of lex expressions

Expression	Matches
(a | b)	a or b
(a | b) (a | b)	aa, ab, ba or bb
abc*	ab, abc, abcc, ...
abc+	abc, abcc, abccc, ...
[abc]	one of : a, b, c
[a – z]	any letter between a and z
[a\ – z]	one of : a, –, z
(ab)*	ε, ab, abab, ...
[\t\n]+	Whitespace
.	Any/ character except newline
\n	Newline
[∧ab]	Anything except : a,b

Some variables are already defined in lex. They can used in writing lex specifications. Here are a few variables :

Name	Functions
int yylex ()	Call to invoke lexer, returns token
char *yytext	Pointer to matching string
yyleng	Length of matching string
yylval	Value associated with token
int yywrap()	yywrap(), return 1 if done, 0 if not done
FILE *yyout	Output file
FILE *yyin	Input file

1.8.1(a) Lex Examples

(1) Give lex specification for prepending line numbers to each line in a file.

```
%{
    int lineno ;
%}
%%
    ∧ (.*) \n        printf ("%d\t%s", ++lineno, yytext) ;
%%
    int main (int arge, char * argv[])
      {
        yyin = fopen (argv [1], "r") ;
        yylex () ;
        fclose (yyin) ;
      }
```

The regular expression ∧(.*)\n defines a line.

(2) Give lex specification for counting number of identifiers in a source file.

```
    digit      [0 – 9]
    letter     [A – Za – z]
%{
    int count ;
%}
%%
    /* match identifiers */
    {letter} ({letter} | {digit})*count ++ ;
%%
int main (int argc, char *argv [])
    {
      yyin = fopen (argv [1], "r") ;
```

Scanned with CamScanner

```
    yylex () ;
    printf ("\n No. of identifiers = %d", count) ;
    return 0 ;
}
```

(3) Give lex specifications for counting the number of characters, words and lines in a file.

```
%{
    int nchar, nword, nline ;
%}
%%
    \n              {nline++ ; nchar ++ ;}
    [^ \t\n]+       {nword++ ; nchar = nchar + yyleng ;}
    .               {nchar ++ ;}
%%
    int main (int arge, char *argv[])
      {
        yyin = fopen (argv [1], "r") ;
        yylex () ;
        printf ("\n%d\t%d\t%d", nchar, nword, nline) ;
        return 0 ;
      }
```

(4) Write lex specifications for recognizing tokens in a subset of a C-Program.

```
delin     [ \t\n]
ws     {delin} +
letter     [A –Za–z]
digit     [0 – 9]
identifier     {letter} ({letter} ! {digit})*
number {digit} + (\. {digit} +) ?
%{
    /* definition of LT, LE, GT, GE, NE, EQ, OPERATOR,
ASSIGN, IF, ELSE, ID, NUMBER, RELOP as constants */
%}
%%
{ws}          {/* no action */}
if            {return (IF) ;}
else              {return (ELSE) ;}
identifier    {yylval = install_id () ; return (ID) ;}
number        {yylval = instal_num () ; return (NUMBER)
;}
"<"               {yylval = LT ; return (RELOP) ;}
"< ="         {yylval = LE ; return (RELOP) ;}
">"               {yylval = GT ; return (RELOP) ;}
">="          {yylval = GE ; return (RELOP) ;}
"!="          {yylval = NE; return (RELOP) ;}
"=="          {yylval = EQ ; return (RELOP) ;}
"="               {yylval = ASSIGN ; return (ASSIGN) ;}
"+"               {yylval = PLUS ; return (OPERATOR)
;}
"–"               {yylval = MINUS ; return (OPERATOR)
;}
%%
```

```
Install_id ()
{
  /*function to store an identifier in the symbol table.
    The symbol is pointed to by yytext.
    The length of the symbol is given by yytext. */
}
Install_num ()
{
  /*function to store a number in the literal table. */
}
```

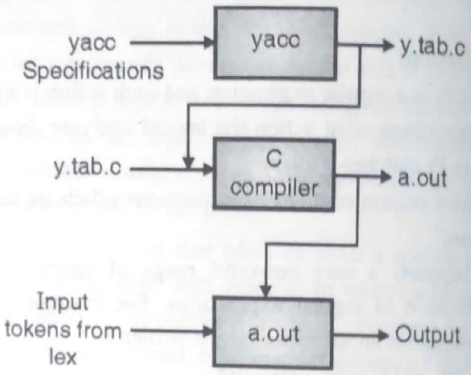
1.8.2 Yacc → (May 2013)

Q. Explain the working of YACC.

SPPU - May 2013, 2 Marks

yacc assists in parsing phase. It creates a C-Program for parser. yacc is available as a command on the UNIX system.

A parser can be constructed using **yacc** in the manner illustrated in Fig. 1.8.2.



(S1.26) **Fig. 1.8.2 : yacc functioning**

- First, a file containing **yacc** specifications for a set of statements is prepared.
- yacc transforms the file containing specifications into a C-Program called y·tab·c, which is a parser in C-Language.
- y·tab·c is run through C-compiler which produces object program a·out.
- The parser a·out can be run to parse a source containing a steam of tokens.

The input of yacc is divided into three sections as given below :

```
declaration
%%
translation rules
%%
C-functions
```

- The declaration section consists of token declarations and C-code bracketed by "%{ and % }".
- The context free grammar is placed in the rule section. Actions can be associated with each rule.
- User functions are added in the last section.

Scanned with CamScanner

The working yacc can be illustrated by constructing a small calculator that can add and subtract numbers. Here is the definition section for the yacc input file.

Definition section for yacc
% token NUMBER

The definition-declaration a **NUMBER** as a token.

- Yacc generates a parser in the file y·tab·c.
- Yacc generates an include file **y·tab·h** for lex. This file associates an integer value with each type of token. The file y·tab·h should be included in lex input file.

To obtain tokens yacc calls **yylex**(). The function **yylex**() returns a token. Value associated with the token is returned by lex in variable **yylval**.

For example, if the rule section of lex specification contains :

```
[0 – 9]+    {yylval = atoi (yytext) ; return (NUMBER) ;}
```

Then the value of the number will be stored in **yylval** and a token **NUMBER** will be returned to yacc by lex.

A rule such as

```
[– +] return *yytext ;
```

Will return – or + to yacc.

The complete lex input specification for our simple calculator is being below.

```
%{
   # include <stdlib.h>
   void yyerror (char *) ;
   # include "y·tab·h"
%}
%%
[0 – 9]+      {yylval = atoi (yytext) ; return (NUMBER) ;}
[– + \n] return *yytext ;
[ \t]   ; /* skip white spaces */
      yyerror ("invalid characters") ;
%%
   int yywrap (void)
     {
       Return 1 ;
     }
```

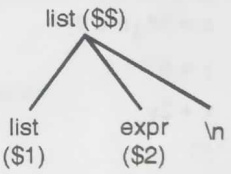
Here is the yacc input specification for our calculator :

```
%{
   # include <stdio.h>
   int yylex (void) ;
   void yyerror (char *) ;
%}
% token NUMBER
%%
   list :
      list expr '\n'          {printf ("%d\n", $2) ;}
      ;
      expr :
      NUMBER                  {$$ = $1 ;}
      ¦ expr '+' expr         {$$ = $1 + $3 ;}
      ¦ expr '–' expr {$$ = $1 – $3 ;}
      ;
%%
void yyerror (char *s)
    {
      printf ("%s \n", s) ;
    }
int main (void)
    {
      yyparse () ;
      return 0 ;
    }
```

- The rule section resembles productions of a context free grammar.
- The left-hand side of a production is entered left-justified and followed by a colon. It is followed by the right-hand side of the production.
- Actions associated with a rule are entered in braces.
- The first rule

  List → list expr '\n' {printf ("%d \n", $2) ;}

  Says that, a program contains a list of expressions.



Each expression terminates by a newline character. The action associated with the above rule says that the evaluated value of the expression, which is in $2 is displayed.

(1) $$ stands for the left hand side variable of a production.

(2) $1 stands for the first symbol on the right hand side of a production.

(3) $i stand for the i^{th} symbol on the right hand side of a production

- The second rule says that, an expression is equal to some of two expressions or difference of two expressions. NUMBER terminates the recursion.

  expr → expr + expr

  lexpr – expr

  lNUMBER

1.8.3 Running Lex and Yacc

Let us assume that the specification files for lex and yacc are lex·l and yacc·y respectively.

Scanned with CamScanner

We need to do the followings in UNIX to build a parser.

```
$ lex lex.l
$ yacc –d yacc.y
$ cc – c lex.yy.c y.tab.c
$ cc – 0 parser lex.yy.0 y.tab.0 – ll
```

- The first line runs lex over lex specification and generates a file, lex.yy.c which contains C code for lexical analysis.
- In the second, we run yacc over yacc specification and generate both y.tab.c and y.tab.h (the latter is the file of token definitions created by the –d switch).
- The next line compiles each of the two programs.
- The final line links them together and uses routines in the lex library.

Example 1.8.1

Implement simple arithmetic operations using lexical analyzer and compiler using **lex** and **yacc.**

Solution :

Our proposed calculator will support following operations :

(1) Add (2) Subtract
(3) Multiply (4) Divide

Parentheses may be used to over-ride operator precedence, and single-character (lower case) variables may be specified in assignment statements. The following illustrates sample user input and the output of the calculator.

```
User :          3* (6 + 9)
Calculator :    45
User :          x = 3* (6 + 9)
User :          y = 6
User :          x + 2y
Calculator :    57
User :          x
Calculator :    45
```

The lexical analyzer returns **ID** and **NUMBER** tokens. For variables **yylval** specifies an index to the symbol table **symtable**. The **symtable** merely holds the value of the associated variable. When **NUMBER** tokens are returned, **yylval** contains the number scanned.

Input specification for lex :

```
%{
   # include <stdio10.h>
   void yyerror (char *) ;
   # include "y.tab.h"
%}
%%
[a – z]   {yylvel = *yytext – 'a' ;
          return ID ;
       }
      /* yylval will contain the index to the symtable. */
[0 – 9] + (\.[0 – 9] + ) ? {yylval = atof (yytext) ; return
                                              NUMBER ;
                          }
[ \t] ;     /* Skip whitespaces */
.       yyerror ("invalid character") ;
%%
   int yywrap(void)
   { return (1) ;
   }
```

Input specifications for yacc

```
% union
   { float dval ;
   int index ;
   }
% token <dval> NUMBER
% token <index> ID
% left '+' '–'
% left '*' '/'
% nonassoc UMINUS
%
{
   void yyerror (char *) ;
   int yylex (void) ;
   float symtable[26] ;
%}
% type <dval> exp
%%
   statement_list : statement '\n'
         | statement_list statment '\n'
      ;
   statement :
         expr          {printf ("%f\n", $1),
         | ID '=' expr     {symtable [$1] = 93 ;}
expr :
   NUMBER
   | ID                                  {$$ = symtable [$1] ;}
   | expr '+' expr          {$$ = $1 + $3 ;}
   | expr '–' expr     {$$ = $1 – $3 ;}
   | expr '*' expr     {$$ = $1 * $3 ;}
   | expr '/' expr          {$$ = $1 / $3 ;}
   | '(' expr ')'           {$$ = $2 ;}
   | '–' expr % prec UMINUS          {$$ = –$2 ;}
   ;
%%
void yyerror (char *s)
{   printf ("%s\n", s) ;
}
int main (void)
{
   yyparse () ;
   return 0 ;
}
```

Scanned with CamScanner

- In the above specification, we have multiple types of symbol value. Expression value is of the type float, while the value of a variable is index in the symbol table.
- To define the possible symbol types, in the definition section we add a % union declaration :

```
% union
    {
      float dval ;
      int index ;
    }
```

- Evaluation of an arithmetic expression has the potential problem of ambiguity. This problem can be handled by telling the yacc about the precedence and associativity of operators.
- yacc lets us specify precedence explicitly. We can add these lines to the definition section :

```
% left '+' '-'
% left '+' '/'
% nonassoc UMINUS
```

Each of these declarations defines a level of precedence. It tells the yacc that "+" and "–" are at the lowest precedence and are left associative. "*" and "/" are at a higher precedence level and are left associative. UMINUS, a pseudo-token standing for unary minus, has no associativity and is at the highest precedence.

1.9 Scanning and Parsing

1.9.1 Scanning → (Dec. 2013)

Q. What is scanning? Briefly explain.
SPPU - Dec. 2013, 4 Marks

The main task of a scanner is to read the input characters and recognize the lexical components. The scanner produces a sequence of tokens to be used by the parser.

- Scanning is separated from parsing to reduce the burden of the parser.
- Lexical components can be specified by regular expressions.
- Regular expressions can be used for automatic generation of DFA. The DFA can be used for recognizing of lexical components.

1.9.2 Parsing

→ (Dec. 2013, May 2015, May 2016)

Q. What is parsing? Briefly explain.
SPPU - Dec. 2013, May 2015, May 2016, 2/4 Marks

We can always find a method for determining whether a particular string is generated by a CFG. Parsing a string is nothing but finding a derivation of the string in the given grammar G. A great deal of work has been done in finding an efficient algorithm for parsing. These algorithms depend on specific properties of the grammar.

PDA is a machine for CFG. A PDA can be used for parsing. A PDA can be enhanced to record its moves, so that the sequence of moves leading to an acceptance of the string can be remembered. Different types of parsing are :

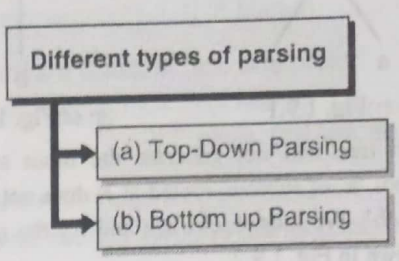


Fig. C1.8: Different types of parsing

→ 1.9.2(a) Top-Down Parsing → (Dec. 2014)

Q. Explain top down parsing with an example.
SPPU - Dec. 2014, 4 Marks

A top down parser for a given grammar G tries to derive a string through a sequence of derivations starting with the start symbol.

A top down parser normally uses leftmost derivation to derive an input string.

- **Recursive-descent parser** is a general top down parser.
- A recursive descent parser may require backtracking and repeated scan of input string. Working of a recursive descent parser is being explained with the help of an example:

☞ Example

Consider the grammar

$S \rightarrow aXb$

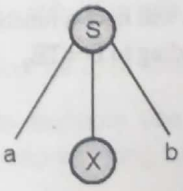
$X \rightarrow ab \mid b$

and the input string abb.

The parse tree of abb can be constructed as given below :

Step 1 : We create a parse tree with single node S. S is the start symbol.

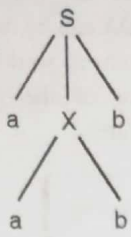
Step 2 : We use the first production $S \rightarrow aXb$ to expand the node



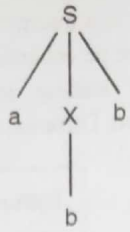
The leftmost leaf, labelled a, matches the first symbol of input string abb. Input pointer is advanced to the next symbol b of abb.

Step 3 : Now, X is expanded using $X \rightarrow ab$ and if it fails to generate remaining input symbols, we backtrack and try the next production $X \rightarrow b$.

Scanned with CamScanner



(S1.51)**Fig. 1.9.1**



(S1.52)**Fig. 1.9.2**

The first tree will not generate the input string abb. The leftmost symbol 'a' of subtree, rooted at X does not match the next input symbol 'b'. We must backtrack and try the next production $X \rightarrow b$, as shown in Fig. 1.9.2.

- A recursive-descent parser may enter an infinite loop.
- We can write a predictive parser (without backtracking) by modifying the grammar :
  (1) Left recursion should be eliminated.
  (2) Apply left factoring to the grammar.
  We can easily write a program for a recursive descent parser.
- There must be a function corresponding to each variable.
- A global variable 'position' points to the current input character.
- If the current input character matches the terminal in the production then the next input character is read by advancing the variable 'position' to the next location.

Example 1.9.1

Write a program for recursive descent parser for the following grammar.

E = E + T | T

T = T * F | F

F = (E) | a | b

Solution :

Step 1 : Removing left recursion from the grammar we get :

$$E = TE_1$$

$$E_1 = +TE_1 \mid \varepsilon$$

$$T = FT_1$$

$$T_1 = *FT_1 \mid \varepsilon$$

$$F = (E) \mid a \mid b$$

Step 2 : The program will have a function for each variable.

Function corresponding to $E = TE_1$

```
E( )
{     T( );
      E1( );
}
```

Function corresponding to $E_1 = +TE_1 \mid \varepsilon$

```
E1( )
{     if (input [position] == '+')
      {     match( );
            T( );
            E1( );
      }
}
```

```
//The function match ( ) will increment the current position
pointer for input string by 1.
```

Function corresponding to $T = FT_1$

```
T( )
{     F( );
      T1( );
}
```

Function corresponding to $T_1 = *FT_1 \mid \varepsilon$

```
T1( )
{     if(input [position] == '*')
      {     match( );
            F( );
            T1( );
      }
}
```

Function corresponding to F = (E) | a | b

```
F( )
{
      if(input[position] == 'C')
      {     match( );
            E ( );
            if (input[position] == ')')
                  match( );
            else
                  error =1;
      }
      else
if(input [position] == 'a'     || input [position] == 'b')
                  match( );
}
// main program
char input [30];
      int error = 0, position = 0;
void main( )
      {
            printf("\n enter a string :");
            gets(input);
            E( );
            if(error == 1 || input [position]! = '\0')
                  printf("\n invalid string");
            else
                  printf("\n valid string");
      }
```

➜ 1.9.2(b) Bottom-Up Parsing

In bottom up parsing, the source string is reduced to the start symbol of the grammar. The bottom up parsing is also known as shift-reduce parsing.

Scanned with CamScanner

- A shift reduce parser constructs a parse tree by beginning at the leaves and then working up towards the root.
- A shift reduce parser can be constructed using a stack.

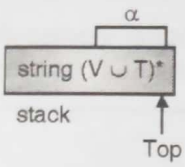
Implementation of shift reduce parser

- A stack is taken to hold grammar symbols.
- An array is taken to store input string to be parsed.
- Initially, the stack is empty.

At each step the parser can take one of the following two steps :

(1) **Reduce :** The process of reduction is shown in the Fig. 1.9.3.

If there is a production $X \rightarrow \alpha$ and the string α is found at the top end of the stack then α should be replaced by X.



(S1.53)**Fig. 1.9.3**

(2) If the reduction step cannot be carried out then the next input symbol is shifted on the stack.

The parser repeats this cycle of shift-reduce until it has detected an error or until the stack contains the start symbol and the input is empty.

Example 1.9.2

Consider the grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

and the input string id + id * id. Show the working of the shift reduce parser.

Solution :

Sr. No.	Stack	Input	Action
1.	empty	id + id *id	Shift
2.	id	+ id * id	Reduce by $F \rightarrow id$
3.	F	+ id * id	Reduce by $T \rightarrow F$
4.	T	+ id * id	Reduce by $E \rightarrow T$
5.	E	+ id * id	Shift
6.	E +	id * id	Shift
7.	E + id	* id	Reduce by $F \rightarrow id$
8.	E + F	* id	Reduce by $T \rightarrow F$
9.	E + T	* id	Shift
10.	E + T*	id	Shift
11.	E + T * id	–	Reduce by $F \rightarrow id$
12.	E + T * F	–	Reduce by $T \rightarrow T * F$
13.	E + T	–	Reduce by
14.	E	–	Accept

1.10 Exam Pack (University Questions)

Q. What is interpreter ? Explain the role of interpreter with suitable example.
(Refer section 1.1.1) (8 Marks) **(Dec. 2013)**

Q. Compare Compiler and Interpreter.
(Refer section 1.1.2) (4 Marks)
(Aug. 2015(In Sem), May 2017)

Q. Explain Compilers and Interpreters.
(Refer section 1.1.2) (6 Marks)**(Oct. 2016(In Sem))**

Syllabus Topic : Components of System Software

Q. Explain program generation activity.
(Refer section 1.2) (4 Marks)**(Oct. 2016(In Sem))**

Q. Explain the steps in program development.
(Refer section 1.2) (7 Marks) **(Dec. 2016)**

Syllabus Topic : Language Processing Activities

Q. What are language processing activities ? Give details of language processing activities.
(Refer section 1.3) (4 Marks)
(Dec. 2013, May 2014)

Q. Explain the Language processing activities.
(Refer section 1.3) (6 Marks)**(Aug. 2015(In Sem))**

Syllabus Topic : Fundamentals of Language Processing

Q. Explain the different phases of language processing.
(Refer section 1.4) (6 Marks) **(Dec. 2015)**

Q. Explain Analysis phase of compiler.
(Refer section 1.4.1) (7 Marks)
(May 2015, May 2016)

Q. Define the term forward reference and explain where it is used with example.
(Refer section 1.4.3) (2 Marks) **(May 2013)**

Q. Explain the terms terminals, non-terminals, starting symbol and production rule set with example.
(Refer section 1.4.3) (6 Marks)**(Oct. 2016(In Sem))**

Q. Explain with example use of Terminals and non-terminals in representing language grammer.
(Refer section 1.4.3) (6 Marks) **(Dec. 2016)**

Q. Define the term DFA and explain where it is used with example.
(Refer section 1.5.1(a)) (2 Marks) **(May 2013)**

Q. Define the term regular expression and explain where it is used with example.
(Refer section 1.5.2) (2 Marks) **(May 2013)**

Scanned with CamScanner

Q. What do you understand by Grammar? Explain the use of Terminal and Non-Terminal in representing grammar.
(Refer section 1.5.3) (4 Marks) **(May 2013)**

Q. What are language processor development tools?
(Refer section 1.8) (4 Marks) **(May 2013)**

Q. Explain different development tools used to develop language processor.
(Refer section 1.8) (6 Marks)
(May 2014, May 2015, May 2016)

Q. Explain the working of Lex.
(Refer section 1.8.1) (2 Marks) **(May 2013)**

Q. Explain the working of YACC.
(Refer section 1.8.2) (2 Marks) **(May 2013)**

Q. What is scanning? Briefly explain.
(Refer section 1.9.1) (4 Marks) **(Dec. 2013)**

Q. What is parsing? Briefly explain.
(Refer section 1.9.2) (2/4 Marks)
(Dec. 2013, May 2015, May 2016)

Q. Explain top down parsing with an example.
(Refer section 1.9.2(a)) (4 Marks) **(Dec. 2014)**

❑❑❑

Scanned with CamScanner

UNIT I

CHAPTER 2

Assemblers

Syllabus Topics

Elements of Assembly language Programming, Simple assembler scheme, Structure of an assembler, Design of single and two pass assemblers.

2.1 Assembly Language Programming

Machine and assembly languages are low level languages since the coding for a problem is at the individual instruction level. An assembly language is machine dependent. It differs from computer to computer.

Writing a program in assembly language is more convenient than in machine language. Instead of binary sequence, as in machine language, it is written in the form of symbolic instructions.

- An assembly program is written using symbols (Mnemonic).
- An assembly program is more readable. It is difficult to understand and develop a program using machine language.
- Assembly language is machine dependent.
- An assembly program is translated into machine code before execution.
- An assembler is a translator which takes its input in the form of an assembly language program and produces machine language code as its output.

(S2.1)**Fig. 2.1.1 : Assembler**

Syllabus Topic : Elements of Assembly Language Programming

2.1.1 Elements of Assembly Language

An assembly language provides three basic features :

1. Operation codes in the form of mnemonics.
2. Symbolic operands.
3. Data declaration.

Let us consider an assembly instruction.

MOV AX, x

- **MOV** is a mnemonic opcode for the operation to be performed.
- **AX** is register operand in a symbolic form
- **X** is a memory operand in a symbolic form.

Let us consider an assembly instruction for data declaration.

X db 5

- Name of the variable is X
- X requires 1 byte of memory
- Initial value of X is set to 5.

The storage specifier **db** reserves one byte of storage for holding a value of X.

☞ Statement format

An instruction in assembly program has the following format :

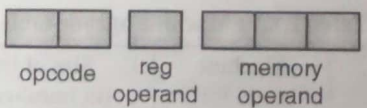
1. **Label :** It is optional, if present it occupies the first field.
2. **Opcode :** It contains the symbolic operation code or storage specification.
3. **Operand :** Operand is a symbolic name for CPU-register or memory variables. There can be a number of operands in an instruction.

[Label] < opcode > < operand > [,< operand >···]

(S2.2)**Fig. 2.1.2 : Format of an assembly instruction**

A general format of an assembly instruction is shown in Fig. 2.1.2. The notation [...] indicates that the enclosed specification is optional.

Operation codes : We will assume a hypothetical assembly language with the machine instruction format shown in Fig. 2.1.3.



(S2.3)**Fig. 2.1.3 : Instruction format**

Our machine has three CPU registers :

1. AREG
2. BREG
3. CREG

Scanned with CamScanner

Our machine supports 11 different operations :

1. STOP – to stop execution
2. ADD – operand1 ← operand1 + operand2
3. SUB – operand1 ← operand1 – operand2
4. MULT – operand1 ← operand1 * operand2
5. MOVER – CPU-register ← memory operand
6. MOVEM – Memory operand ← CPU-register
7. COMP – Set condition code, these condition codes will be used by the conditional branch instruction **BC**.
8. BC – Branch on condition. Conditions to be used for branching are :
   a) EQ – equal
   b) NE – not equal
   c) LT – less than
   d) GT – greater than
   e) LE – less or equal
   f) GE – greater or equal
   g) ANY
9. DIV – operand1 ← operand1 / operand2
10. READ – operand2 ← input value
11. PRINT – output ← operand2

- First operand is always a CPU register
- Second operand is always a memory operand
- READ and PRINT instructions do not use the first operand
- The stop instruction has no operand.

Example : Meaning of some typical instructions are give below :

Statement	Meaning
ADD AREG, X	AREG ← AREG + X
MOVER BREG, X	BREG ← X
MOVEM CREG, X	X ← CREG
COMP AREG, Y	Compare AREG and the memory variable Y and set the necessary condition codes. These condition codes are required by **BC** instruction.

Each symbolic opcode is associated with machine opcode. These details are stored in machine opcode table (MOT).

A MOT contains :

1. Opcode in mnemonic form
2. Machine code associated with the opcode.

MOT for our hypothetical assembly language is given in Table 2.1.1.

Table 2.1.1 : Machine opcode table

Symbolic opcode (mnemonic)	Machine code for opcode	Size of instruction (in number of words)
STOP	00	1
ADD	01	1
SUB	02	1
MULT	03	1
MOVER	04	1
MOVEM	05	1
COMP	06	1
BC	07	1
DIV	08	1
READ	09	1
PRINT	10	1

- Machine opcode is a two digit code.
- Size of each instruction is assumed to be of 1 word.

Example 2.1.1

In certain cases assembly language programming holds an edge over high level language programming. Comment on the above statement.

Solution :

Assembly language is machine dependent. A machine dependent application cannot be developed using a high level language. Such applications are best developed in assembly language.

A program written in assembly language is more efficient than a program written is high level language. A compiler cannot efficiently utilize various architectural features of a machine. In addition, the code generated by a compiler is not very efficient and optimized.

2.1.2 Assembly Program to Machine Language Program

An assembly program can be translated into machine language. It involves following steps :

1. Find addresses of variables and labels.
2. Replace symbolic addresses by numeric addresses.
3. Replace symbolic opcodes by machine operation codes.
4. Reserve storage for data.

Let us consider a sample program as shown in Fig. 2.1.4. The program finds the sum of two numbers X and Y.

```
          START   101
          READ    X
          READ    Y
          MOVER   AREG, X
          ADD     AREG, Y
          MOVEM   AREG, RESULT
          PRINT   RESULT
          STOP
X         DS  1
Y         DS  1
RESULT    DS  1
          END
```

Fig. 2.1.4 : A sample program for finding X + Y

Generation of machine code for program of Fig. 2.1.4 is being explained, step-wise.

Scanned with CamScanner

Step 1 : Finding addresses of variables :

The instruction, 'START 101' is an assembler directive. This directive indicates that the address (location count or LC) of the first instruction will be 101.

- LC value of the first executable instruction will be 101.
- LC value is incremented by 1 after every instruction as each machine instruction requires a word of memory.

The instruction 'X DS 1' is for reserving one word of memory for the variable X. The DS stands for declare storage.

The program after LC processing is shown in Fig. 2.1.5.

Location counter (LC)	Assembly program		
		START	101
101		READ	X
102		READ	Y
103		MOVER	AREG, X
104		ADD	AREG, Y
105		MOVEM	AREG, RESULT
106		PRINT	RESULT
107		STOP	
108	X	DS	1
109	Y	DS	1
110	RESULT	DS	1
		END	

Fig. 2.1.5 : Program after LC processing

After LC processing, the addresses of the three variables are given below :

Variable	Address
X	108
Y	109
RESULT	110

Step 2 : Generation of machine code

LC	Assembly Instruction	Machine code
101	READ X	09 0 108 — 09: opcode for READ as declared in table 2.1.1; 0: Absence of register operand is shown by 0; 108: Address of X
102	READ Y	09 0 109 — 09: opcode for READ; 0: Register operand is missing; 109: address of Y
103	MOVER AREG, X	04 1 108 — 04: opcode for MOVER; 1: Stands for AREG; 108: address of X
104	ADD AREG, Y	01 1 109 — 01: opcode for ADD; 1: Stands for AREG; 109: address of Y
105	MOVEM AREG, RESULT	05 1 110 — 05: opcode for MOVEM; 1: Stands for AREG; 110: address of RESULT
106	PRINT RESULT	10 0 110 — 10: opcode for PRINT; 0: Absence for register; 110: address of RESULT
107	STOP	00 0 000 — 00: opcode for STOP; 0: Absence for register; 000: address of memory operand
108	X DS 1	Memory is reserved but no-code is generated.
109	Y DS 1	
110	RESULT DS 1	

Thus the required machine code will be :

LC	Opcode	Register	Address
101	09	0	108
102	09	0	109

Scanned with CamScanner

LC	Opcode	Register	Address
103	04	1	108
104	01	1	109
105	05	1	110
106	10	0	110
107	00	0	000
108			
109			
110			

2.1.3 Assembly Language Statements

→ (Dec. 2016, May 2017)

Q. Explain different assembly language statements with examples. **SPPU-Dec. 2016, 7 Marks**

Q. Explain in brief imperative statements, declaration statements and assembler directives with examples for assembly language programming. **SPPU-May 2017, 9 Marks**

There are three types of statements in an assembly program. These are :

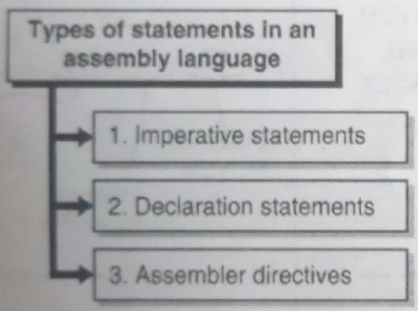


Fig. C2.1: Types of statements in an assembly language

→ 1 Imperative statements

These are executable statements. Each imperative statement indicates an action to be taken during execution of the program. Each imperative statement translates into a machine instruction.

Example : Some sample imperative statements are given below :

```
1.  MOVER   BREG, X
2.  STOP
3.  READ    X
4.  PRINT   Y
5.  ADD     AREG, Z
6.  BC      NE, L1
```

→ 2. Declaration statements

Declaration statements are for reserving memory for variables. We can specify the initial value of a variable. Syntax of declaration statements is as follows :

[Label] DS <Constant specifying the size of memory to be reserved>

[Label] DC '<Value specifying the initial value of the variable>'

DS : DS stands for declare storage

DC : DC stands for declare constant

DS statement can be used for reserving a block of memory for a variable

```
X   DS  1
Y   DS  5
```

The first statement 'X DS 1' reserves a memory area of 1 word for the variable X. The second statement 'Y DS 5' reserves a memory area of 5 words for the variable Y.

DC statement can be used for reserving 1 word of memory for a variable. Initial value of the variable can be specified.

```
ONE DC  '1'
```

The statement 'ONE DC '1' ' does the following :

1. Reserves one word of memory for the variable ONE.
2. Memory is initialized with '1'.

→ 3. Assembler directives

→ (May 2016)

Q. Explain in brief assembler directives with examples. **SPPU - May 2016, 6 Marks**

Assembler directives instruct the assembler to perform certain actions during assembly of a program.

- A directive is a direction for the assembler.
- A directive is also known as pseudo instruction.
- Normally, machine code is not generated for assembler directives.

Some assembler directives are :

```
START   <constant>
END     [<operand specification>]
```

The **START** directive indicates that the first word of the machine code should be placed in the memory word with address <constant>

The **END** directive indicates the end of the source program. The optional <operand specification> indicates the address of the instruction where the execution of the program should begin. By default, the execution begins with the first instruction.

2.1.4 Literals and Constants

A literal is an immediate operand appearing in a statement.

In the C-statement

```
x = y + 5;
```

The constant value '5' is known as literal.

In the C-statement

```
int Z = 5 ;
```

The value '5' is a constant. Initial value of Z is set to 5. Z can be modified in subsequent instructions.

- A literal is an operand with constant value.
- A literal can not be changed during program execution.
- A literal is more safe and protected than a constant.
- Literals appear as part of the instruction.

They are specified using immediate addressing.

Scanned with CamScanner

Literals in assembly language

Many machines support immediate operands in machine instruction. Assembly language of 8086 supports this feature. We can write an assembly instruction for 8086 with immediate operand.

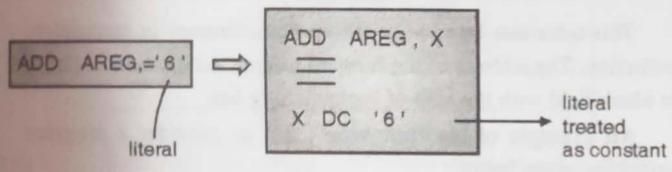
MOV AX, 15 [8086 instruction]

Our, hypothetical machine does not support immediate operand as part of the machine instruction. Our, simple machine can still handle literals. The process of handling a literal by our machine is being explained below :

When the assembler finds a literal in the operand field of a statement :

(a) It allocates memory word to the literal. The value of the literal is stored in the allocated memory word.

(b) It then replaces the literal in the statement with the address of the allocated memory word.



(S2.5)**Fig. 2.1.6 : Handling of literals**

Handling of literal is shown in Fig. 2.1.6.

- The literal is stored as a constant at the memory location referred to by X.
- The immediate operand 6 in the statement ADD AREG, 6 is replaced by the address of X.

ADD AREG, X

The technique of handling literals as explained above should be used only in those cases, where the machine does not support immediate operands and still we want to have literals in assembly statements.

Example 2.1.2

Immediate operands and literals are both ways of specifying an operand value in a source statement. What are the merits and demerits of each ? When might each be preferable to the other ?

Solution :

- A machine is required to support immediate operands. Many machines provide support for immediate addressing. In such machines, an immediate operand can be used instead of literals.
- Literals need not be supported by machine architecture. When the assembler encounters the use of literal in operand field of a statement, it allocates memory word to store the value of the literal and replaces literal in the statement by its address.
- An instruction using immediate operand executes faster as the operand is part of the instruction. An additional memory fetch is not needed to get the operand value from the memory.
- An assembly instruction with a literal will ultimately use direct addressing. Direct addressing requires an additional memory fetch to bring the value of the operand.

2.2 Assembler and Related Programs

→ (Dec. 2013)

Q. What is an assembler ? SPPU - Dec. 2013, 4 Marks

An assembler translates an assembly program and generates information necessary for running the program. The translated program contains three kinds of entities :

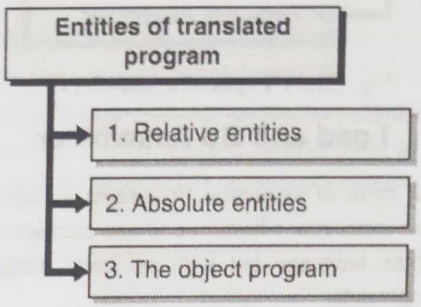


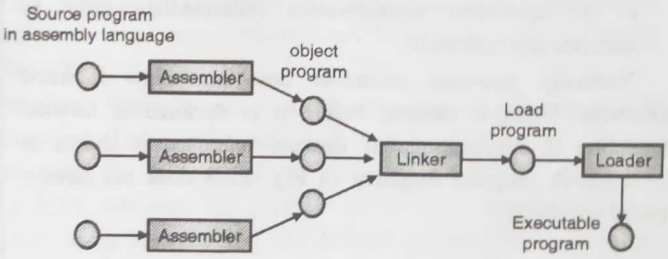
Fig. C2.2: Entities of translated program

→ 1. Relative entities

Relative entities include addresses of instructions and variables. These are fixed with respect to each other and are stated relative to the beginning of the module.

→ 2. Absolute entities

Absolute entities include operation codes, numeric and string constants and fixed addresses. The values of absolute entities are independent of which storage locations the resulting machine code will eventually occupy.



(S2.6)**Fig. 2.2.1 : Program translation**

→ 3. The object program

The object program includes identification of :

- which addresses are relative
- which symbols are defined externally.
- which symbols are defined internally and referenced externally.

- The external references are resolved for two or more modules by a linker. The linker accepts the several object programs as input and produces a single program ready for loading.
- The output of a linker is input for the loader. The loader readjusts the relative entities in a program and then loads the program in memory for execution.

Scanned with CamScanner

There are three types of assemblers :

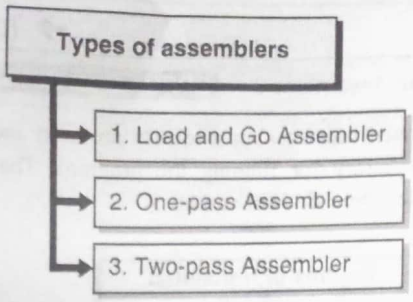


Fig. C2.3 : Types of assemblers

➜ 2.2.1 Load and Go Assembler

It is a simplest form of assembler. It accepts as input assembly programs whose instructions have one to one correspondence with those of machine language but with symbolic names used for operators and operands.

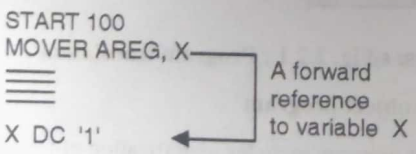
- It produces machine language as output which are loaded directly in main memory and executed.
- The load and go assembler has the ability to design code and test different program components in parallel.
- Change in one particular module does not require scanning the rest of program.

➜ 2.2.2 One-pass Assembler

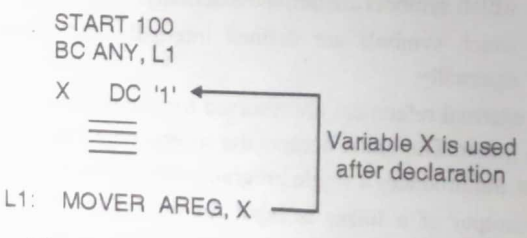
A pass of an assembler is scan of the source program or its equivalent representation.

- A one-pass assembler requires 1 scan of the source program to generate machine code.
- A two-pass assembler requires 2 scans of the source program or its equivalent representation (intermediate code) to generate machine code.

Normally, one-pass assembler does not allow **forward references**. Using a variable before it is declared is forward referencing it. An example of forward reference is shown in Fig. 2.2.2. A program fragment in Fig. 2.2.3 does not have a forward reference.



(S2.7)**Fig. 2.2.2 : An example of forward reference**



(S2.8)**Fig. 2.2.3 : A program with backward reference**

- An assembler cannot generate machine code for an assembly instruction with forward references.
- Machine code for an assembly instruction can be generated, after the address of the variable used in the instruction is known.
- A data structure, **symbol table** is used to record addresses of variables. These addresses can be used during generation of machine code.

2.2.2(a) One-pass Assembler with Forward References

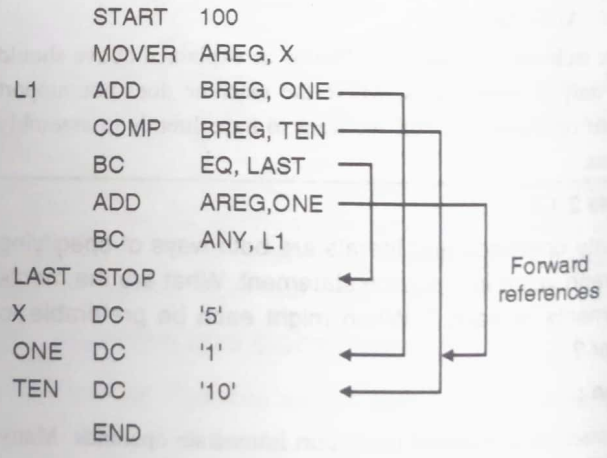
The problem of forward references can be tackled using a technique known as **backpatching**.

- The operand field of an instruction containing forward references is left blank initially.
- A table of instruction containing forward reference is maintained separately. An entry is this table is a pair.
  (<instruction address>, <symbol making a forward reference>)

This table can be used to fill-up the addresses in incomplete instruction. The address of the forward referenced symbols is put in the blank field with the help of backpatching list.

An example of backpatching : Let us consider a program segment as given below :

- The instruction 'L1 ADD BREG, ONE' makes a forward reference to the variable ONE.
- The instruction 'COMP BREG, TEN' makes a forward to the variable TEN.
- The instruction 'BC EQ, LAST' makes a forward reference to the label 'LAST '.



Step 1 : Generation of machine code with operand field containing a forward reference, left as blank.

Assembly instruction			Machine instruction			
	START	100				
	MOVER	AREG, X	100	04	1	--- To be backpatched
L1	ADD	BREG, ONE	101	01	2	---
	COMP	BREG, TEN	102	06	2	---

Scanned with CamScanner

Assembly instruction			Machine instruction			
	BC	EQ, LAST	103	07	1	---
	ADD	AREG, ONE	104	01	1	---
	BC	ANY, L1	105	07	7	101
LAST	STOP		106	00	0	000
X	DC	'5'	107	00	0	005
ONE	DC	'1'	108	00	0	001
TEN	DC	'10'	109	00	0	010

Backpatch list

Instruction Address	Symbol making a forward reference
100	X
101	ONE
102	TEN
103	LAST
104	ONE

Step 2 : Backpatch list can be used to fill up addresses in blank fields of instructions.

Machine instructions after backpatching :

```
100  04  1  107
101  01  2  101
102  06  2  109
103  07  1  106
104  01  1  108
105  07  7  101
106  00  0  000
107  00  0  005
108  00  0  001
109  00  0  010
```

Syllabus Topic : Design of Single Pass Assembler

2.2.2(b) Design of Single Pass Assembler

Data structures needed

1. Op code table
2. Symbol table
3. Backpatch list
4. Literal table

Algorithm

```
LC = 0; // Location counter loop until the end of the program
{
    1. Read in a line of assembly code.
    a. If there is label then insert the label with LC in the symbol table.
    b. If assembler directive then process it
    c. If executable instruction then generate machine code. If the instruction contains a symbol with forward reference then enter the same in the Backpatch list.
}
Process the Backpatch list.
```

Syllabus Topics : Simple Assembler Scheme, Structure of an Assembler

2.2.3 Two Pass Assembler

Mostly assemblers are designed in two passes. The pass-wise grouping of tasks in a two pass assembler is given below :

Pass I

1. Separate the labels, mnemonic op-code and operand fields.
2. Determine the storage requirement for every assembly language statement and update the location counter.
3. Build the symbol table. Symbol table is used to store each label and each variable and its corresponding address.

Pass II : Generate machine code.

Example 2.2.1

Why must the one pass assembler for go literals ?

Solution : A two pass assembler can handle literals.

In the first pass

When the assembler encounters the use of literal in operand field of a statement, it allocates memory word to store the value of the literal. This is done with the help of literal table.

In the second pass

The assembler replaces every literal by its address.

A one-pass assembler will not be able to assign memory address to a literal. Hence, one-pass assembler will not be able to handle literals.

Example 2.2.2

What features of assembly language makes it mandatory to design a two pass assembler ? Explain with suitable example.

Solution : Normally, one-pass assembler does not allow **forward references**. Using a variable before it is declared is forward referencing it. An example of forward reference is shown in Fig. 2.2.2. Although the problem of forward references can be tackled using backpatching, the process of backpatching is quite difficult.

Literals cannot be handled by one-pass assembler.

Thus, forward references and use of literals make it mandatory to design a two pass assembler.

Example 2.2.3

The process of fixing up a few forward references should involve less overhead than making a complete second pass of the source program. Why don't all assemblers use the one-pass technique for efficiency ?

Solution : Normally, one-pass assembler does not allow forward references. The problem of forward references can be tackled using backpatching.

- A one pass assembler cannot handle literals.
- A one pass assembler cannot handle statements like.

```
X    EQU    Y + 10
```

where Y is a forward reference.

Scanned with CamScanner

Backpatching in every case could become very difficult. A two pass assembler can easily handle every type of forward reference including literals.

- The speed of second pass can be improved by using intermediate code. Therefore, it is best to use a two pass assembler with intermediate code.

2.3 Advanced Assembler Directives

In addition to basic assembler directives, we can have some advanced directives. These directives include.

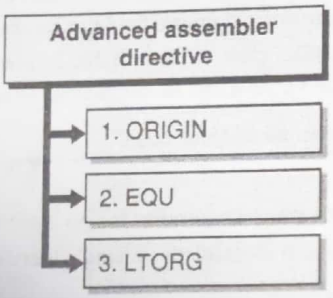


Fig. C2.4 : Advanced assembler directive

➔ 1. ORIGIN

➔ **(May 2014, Aug. 2015, Oct. 2016)**

Q. Explain ORIGIN statement with example.
SPPU - May 2014, Aug. 2015(In Sem), Oct. 2016(In Sem), 2 Marks

The syntax of this directive is :

ORIGIN <address specification>

where <address specification> is an operand, a constant or an expression containing an operand and a constant.

- This directive sets the address of LC to the address given by <address specification>.
- The ORIGIN directive is useful when the machine code is not stored in consecutive memory locations.
- This directive gives the ability to perform LC processing in a relative rather than absolute manner.

Example

Let us consider the assembly program given in Fig. 2.3.1.

Sr. No.	Assembly program			LC
1		START	100	
2		MOVER	BREG,='2'	100
3	LOOP	MOVER	AREG, N	101
4		ADD	BREG,='1'	102
5		ORIGIN	LOOP+5	
6	NEXT	BC	ANY, LOOP	106
7		LTORG		
8		ORIGIN	NEXT+2	
9	LAST	STOP		108
10	N	DC	'5'	109
11		END		

Fig. 2.3.1 : An assembly program illustrating ORIGIN

- The statement number 5, viz. ORIGIN LOOP+5, sets LC to the value 106. The label LOOP is associated with the address 101.
- The statement number 8, viz. ORIGIN NEXT+2, sets LC to the value 108. The label NEXT is associated with the address 106.

➔ 2. EQU

➔ **(May 2014, Aug. 2015)**

Q. Explain EQU statement.
SPPU - May 2014, Aug. 2015(In Sem), 2 Marks

The EQU statement has the syntax

<symbol> EQU <address specification>

where <address specification> can be an <operand specification> or a <constant>

The EQU simply associates the <symbol> with the <address specification>

e.g.

BACK EQU LOOP

The symbol BACK is set to the address of LOOP.

Example 2.3.1

An assembly language program contains the statement : XEQU Y + 25. Indicate how the EQU statement can be processed if (i) Y is back reference (ii) Y is forward reference

Solution :

(i) Y is back reference

The EQU statement will associate the symbol X with the address Y + 25. The assembler will store the symbol X with the address Y + 25 in the symbol table. This can be handled in pass I.

(ii) Y is forward reference

If Y is a forward reference then its address cannot be fixed at the time the assembler finds the statement

X EQU Y + 25

The assembler can generate an equivalent intermediate code for 'Y + 25' in Pass I and in Pass II, the address of X can be filled up.

➔ 3. LTORG

➔ **(May 2014, Oct. 2016)**

Q. What is a LTORG Statement ? Explain with example.
SPPU - May 2014, Oct. 2016(In Sem), 8 Marks

The LTORG statement permits a programmer to specify where literals should be placed. If the LTORG statement is not present, literals are placed after the END statement.

At every LTORG statement, memory is allocated to the literals of the current pool of literals. The pool contains all literals used in the program since the start of the program or since the last LTORG statement.

Scanned with CamScanner

Syllabus Topic : Design of Two Pass Assembler

2.4 Design of Two Pass Assembler

→ **(May 2013, Dec. 2015)**

Q. How Pass I of an assembler works.
SPPU - May 2013, 4 Marks

Q. Explain two pass assembler along with the schematic diagram. **SPPU - May 2013, 4 Marks**

Q. Describe the design of Pass 1 of two pass assembler.
SPPU - Dec. 2015, 7 Marks

An assembler is a program that accepts an assembly language program as input and produces its machine code. An assembler has to perform certain functions in translating from assembly language to machine code. These functions are :

1. Replace symbolic address by numeric address.
2. Replace symbolic operation codes by machine operation codes.
3. Reserve storage for instruction and data
4. Handle literals.

Because variables can be used before they are defined, it is convenient to make a two pass assembler. First pass has to fix addresses of variables; the second pass can generate the machine code.

Consider the assembly statement

ADD AREG, X

Following information is needed to produce machine instruction corresponding to this statement.

1. Machine operation code for the mnemonic ADD.
2. Address of the variable 'X'.

Assembler will need the following data structures during processing.

1. Machine operation code table (MOT)
2. Symbol table (ST).

Each entry in MOT has two primary fields :

1. Mnemonic 2. Opcode

MOT is used during generation of machine code. Assembler can obtain machine opcode corresponding to a mnemonic.

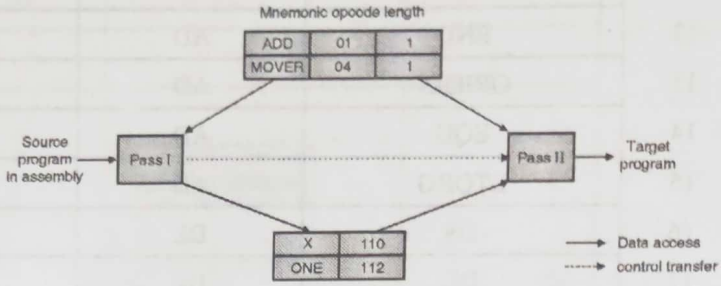
Each entry in ST has two primary fields.

1. Name 2. Address

The symbol table (ST) is built in first pass. During second pass, the assembler can obtain addresses of variables and labels from the symbol table.

Fig. 2.4.1 illustrates the use of data structures in two passes.

- As symbols (variables and labels) are handled using a symbol table (ST), literal are handled using a literal table (LT). Each entry in literal table (LT) has two primary fields :
  1. Value of the literal
  2. Address of the associated memory location.



(S2.10) **Fig. 2.4.1 : Data structures for the assembler**

- In the first pass, literal table (LT) is used to collect literals. LTORG directive permits a programmer to specify where the literals should be placed in the memory. Subsequent literals in the program are placed in the memory on encountering the next LTORG statement. Thus, literals are divided into pools.
- These pools of literals are maintained using an auxiliary pool table (PT). This table contains the literal number of starting literal of each literal pool.

  Pool table (PT) has one field :

  1. Starting literal number of each pool.

  On encountering an LTORG statement (or the END statement), literals in the current pool are allocated addresses starting with the current value in LC and LC is appropriately incremented.

☞ Working of pass I of a two pass assembler

Pass I uses the following data structures :

1. Machine operation code table (MOT)
2. Symbol table (ST)
3. Literal table (LT)
4. Pool table (PT).

Contents of MOT are fixed for an assembler. Sample contents of MOT are shown in Fig. 2.4.2. For convenience mnemonic opcodes, psuedo opcodes, names of registers and comparison conditions are clubbed in a single table.

Table 2.4.1 : An enhanced machine opcode table (MOT)

	Mnemonic opcode	Class	Opcode	Length
0	STOP	IS	00	1
1	ADD	IS	01	1
2	SUB	IS	02	1
3	MULT	IS	03	1
4	MOVER	IS	04	1
5	MOVEM	IS	05	1

IS : Imperative Statement
AD : Assembler Directive
DL : Declaration Statement
RG : Register

Scanned with CamScanner

	Mnemonic opcode	Class	Opcode	Length
6	COMP	IS	06	1
7	BC	IS	07	1
8	DIV	IS	08	1
9	READ	IS	09	1
10	PRINT	IS	10	1
11	START	AD	01	–
12	END	AD	02	–
13	ORIGIN	AD	03	–
14	EQU	AD	04	–
15	LTORG	AD	05	–
16	DS	DL	01	–
17	DC	DL	02	1
18	AREG	RG	01	–
19	BREG	RG	02	–
20	CREG	RG	03	–
21	EQ	CC	01	–
22	LT	CC	02	–
23	GT	CC	03	–
24	LE	CC	04	–
25	GE	CC	05	–
26	NE	CC	06	–
27	ANY	CC	07	–

CC : Comparison Conditio

Contents of various tables for processing of program of Fig. 2.4.2 are being listed after every program instruction.

```
        START   200
        MOVER   AREG, ='5'
        MOVEM   AREG, X
L1      MOVER   BREG, ='2'
        ORIGIN  L1+3
        LTORG
NEXT    ADD     AREG,='1'
        SUB     BREG,='2'
        BC      LT,BACK
        LTORG
BACK    EQU     L1
        ORIGIN  NEXT + 5
        MULT    CREG, 4
        STOP
X       DS      1
        END
```

Fig. 2.4.2 : An assembly program for understanding pass I

Scanned with CamScanner

Sr. No.	LC	Assembly statement	Symbol table	Literal table	Pool table
		Initially	STN→ 0 Symbol | Address	0 ← LTN Literal | Address	Literal No. 0 ← PTN
1.	–	START 200			
2.	200	MOVER AREG, = '5'		0 = '5' – 1 ← LTN Literal | Address	
3	201	MOVEM AREG, X	0 X – 1 ←STN Symbol | Address		
4.	202	L1 MOVER BREG, = '2'	0 X – 1 L1 202 2 ←STN Symbol | Address	0 = '5' – 1 = '2' – 2 ← LTN Literal | Address	
5.		ORIGN L1 + 3			
6.	205 206	L TORG = '5' = '2'		0 = '5' 205 1 = '2' 206 Literal | Address	Literal No. 0 2

Scanned with CamScanner

Sr. No.	LC	Assembly statement	Symbol table	Literal table	Pool table
7.	207	NEXT ADD AREG, = '1'	0 X 1 L1 202 2 NEXT 207 3	0 = '5' 205 1 = '2' 206 2 = '1' – 3	

2.4.1 Intermediate Code

Intermediate code is an equivalent representation of source program.

- Pass I of the assembler involve scanning of the source file. It has to separate labels, mnemonics, op-code and operand fields.
- Every opcode is searched in the machine opcode table.
- Every operand (variable) is searched in the symbol table.

In intermediate code, processed information is stored. It helps in avoiding :

1. Scanning of source file in pass II.
2. Searching MOT and ST in pass II.

Thus, intermediate code increases the processing efficiency in pass II.

Format of intermediate code

For every line of assembly statement, one line of intermediate code (IC) is generated.

- Each mnemonic opcode field is represented as
  (Statement class, machine code)
- The statement class can be :
  1. IS : Imperative statement
  2. DL : Declaration statement
  3. AD : Assembler directive
- The machine opcode for mnemonic code is shown in Table 2.4.1.
- For example :
  1. The mnemonic opcode MOVER will be represented as (IS, 04)
  2. The pseudo mnemonic opcode LTORG will be represented as (AD, 05).
  3. The assembler directive START will be represented as (AD, 01).
  4. DC will be represented as (DL, 02).
- Each operand field is represented as
  (Operand class, reference)
- The operand class can be :
  C : Constant
  S : Symbol (variable)
  L : Literal
  RG : Register
  CC : Condition code
- For a symbol or literal, the reference field contains the index of the operand's entry in symbol table or literal table.
- An example of intermediate code is shown in Fig. 2.4.3.
- For a constant, the reference field contains the constant itself.

		Assembly program	LC	Intermediate code
1.		START 200		(AD,01) (C,200)
2.		MOVER AREG, = '5'	200	(IS,04) (RG, 01) (L, 0)
3.		MOVEM AREG, X	201	(IS,05) (RG, 01) (S, 0)
4.	L1	MOVER BREG, = '2'	202	(IS, 04) (RG, 02) (L,1)
5.		ORIGIN L1 + 3	203	(AD, 03) (C, 205)
6.		LTORG	205	(DL, 02) (C, 5)
			206	(DL, 02) (C, 2)
7.	NEXT	ADD AREG, = '1'	207	(IS, 01) (RG, 01) (L, 2)
8.		SUB BREG, = '2'	208	(IS, 02) (RG, 02) (L, 3)
9.		BC LT, BACK	209	(IS, 07) (CC, 02) (S, 3)
10.		LTORG	210	(DL, 02) (C, 1)
			211	(DL, 02) (C, 2)
11.	BACK	EQU L1	212	(AD, 04) (C, 202)
12.		ORIGIN NEXT + 5	212	(AD, 03) (C, 212)
13.		MULT CREG, = '4'	212	(IS, 03) (RG, 03) (L, 4)
14.		STOP	213	(IS, 00)
15.	X	DS 1	214	(AD, 02)
16.		END	215	(AD, 02)
			215	(DL, 02) (C, 4)

Symbol Table

	Symbol	Address
0	X	214
1	L1	202
2	NEXT	207
3	BACK	202

Literal Table

	Literal	Address
0	= '5'	205
1	= '2'	206
2	= '1'	210
3	= '2'	211
4	= '4'	215

Pool Table

0	0
1	2
2	4

Fig. 2.4.3 : An example of intermediate code

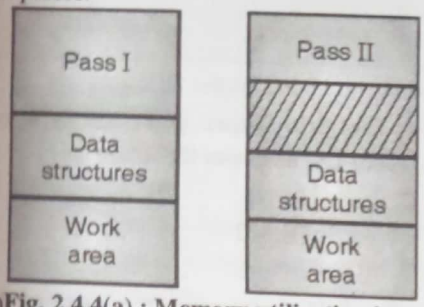
Scanned with CamScanner

2.4.1(a) Variants of Intermediate Code

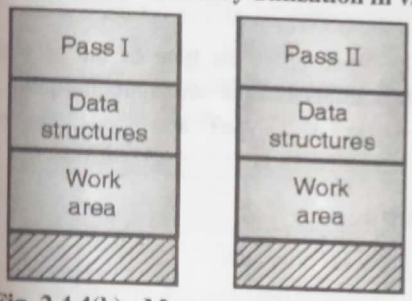
There could be many variants of intermediate code. We will confine ourselves to two variants.

1. Variant I : Operands are processed in pass I.
2. Variant II : Operands are processed in pass II.

– Variant I does more work in pass I. Operand fields are completely processed in pass I. The task of pass II becomes quite simple. Intermediate code becomes quite compact. Memory and processing requirements are higher in pass I. Fig. 2.4.4 (a) shows memory utilization by an assembler in different passes.



(S2.11(a)) **Fig. 2.4.4(a) : Memory utilization in variant I**



(S2.11(b)) **Fig. 2.4.4(b) : Memory utilization in variant II**

– Variant II reduces the work of pass I by transferring the work of operand processing from pass I to pass II. In variant II, pass II of the assembler has to do more work. Here the processing requirement is evenly distributed over two passes. Code sizes of two passes are comparable (as shown in Fig. 2.4.4(b)). In variant II, the overall memory requirement of the assembler is lower.

Variant I

In variant I, each operand is represented by a pair of the form (operand class, code)

The operand **class** is one of :

1. S for symbol
2. C for constant
3. L for literal
4. RG for register
5. CC for condition code

The value of **code** is :

1. Index of the operand entry in the symbol table for a symbol (variable, label).
2. Index of the operand entry in the literal table for a literal.
3. Internal representation of a constant for a constant as an operand.
4. Serial number of the register for a CPU register.
5. Serial number of the condition code for a condition code as operand.

A sample intermediate code using variant I is shown in Fig. 2.4.5.

	START	100	(AD, 01) (C, 100)
L1	READ	A	(IS, 09) (S, 1)
	SUB	AREG, = '5'	(IS, 02) (RG, 01) (L, 0)
	BC	GT, L1	(IS, 07) (CC, 03) (S, 0)
	STOP		(IS, 00)
A	DS	1	(DL, 01) (C, 1)
	–		–
	–		–
	–		–

Fig. 2.4.5 : Intermediate code using variant I

Variant II

In variant II, operands are processed selectively.

1. Constants are processed.
2. Literals are processed.
3. Symbols, condition codes and CPU registers are not processed.

A sample intermediate code using variant II is shown in Fig. 2.4.6.

	START	100	(AD, 01) (C, 100)
L1	READ	A	(IS, 09) A
	SUB	AREG, = '5'	(SI, 02) AREG, (L, 0)
	BC	GT, L1	(IS, 07) GT, L1
	STOP		(SI, 00)
A	DS	1	(DL, 01) (C, 1)
	–		–
	–		–
	–		–

Fig. 2.4.6 : Intermediate code using variant II

2.4.2 Algorithm for Pass I

→ **(May 2013)**

Q. Explain the algorithm in detail with data structures used.
SPPU- May 2013, 4 Marks

Data structures for pass I

Pass I uses the following data structures :

1. Machine opcode table (MOT)
2. Symbol table (ST)
3. Literal table (LT)
4. Pool table (PT)
5. Table for storing intermediate code in internal format (IC)

1. Structure of machine opcode table (MOT)

```
struct MOTtable
{
    char Mnemonic [6] ;
    int Class ;
    char Opcode [3] ;
};
```

Scanned with CamScanner

The class field indicates the following :

Sr. No.	Type of mnemonic symbol	Value of class field
1.	Imperative statement (IS)	1
2.	Declaration statement (DL)	2
3.	Assembler directive (AD)	3
4.	CPU register (RG)	4
5.	Condition code (CC)	5

Initial value of MOT is given below. It is a fixed table and it does not change during the course of program execution.

```
struct MOTtable MOT [28] =
{{"STOP",1, "00"},{"ADD",1, "01"},{"SUB",1, "02"},
{"MULT",1, "03"}, {"MOVER",1, "04"},
"MOVEM", 1, "05"},
"COMP",1,"06"},{"BC",1, "07"}, {"DIV", 1, "08"},
{"READ",1, "09"},{"PRINT",1, "10"},
{"START", 3, "01"},{"END",3, "02"},
{"ORIGIN",3, "03"},
{"EQU",3, "04"},{"LTORG",3,"05"},
{"DS",2, "01"},{"DC",2, "02"},
{"AREG",4, "01"},{"BREG",4, "02"},
{"CREG",4, "03"},
{"EQ",5, "01"},{"LT",5, "02"}, {"GT", 5, "03"},
"LE",5, "04"},
{"GE",5, "05"}, {"NE",5, "06"}, {"ANY",5, "07"}} ;
```

2. Symbol table (ST)

```
struct symboltable
{
    char symbol [8] ;
    int address ;
    int size ;
}ST[20] ;
```

A symbol table entry contains the followings :

1. Name of a variable or a label.
2. Its address.
3. Its size in number of words.

3. Literal table (LT)

```
struct literaltable
{
    int literal ;
    int address ;
}LT[10] ;
```

A literal table entry contains the following:

1. Value of the literal.
2. Address of the memory location associated with the literal. Literals are accommodated in memory on seeing the 'LTORG ' statement or after the 'END' statement.

4. Pool table (PT)

```
int PT[10] ;
```

This table contains the literal number of the starting literal of each literal-pool.

PT[0] : Gives the literal number (index of literal table) of pool 0.

PT[i] : Gives the literal number (index of literal table) of pool i.

5. Intermediate code table (IC) :

```
struct intermediatecode
{
    int LC ;
    int Code1,Type1;
    int Code2,Type2;
    int Code3,Type3;
}IC[30];
```

An IC table entry contains the followings :

1. **Type 1 :** The field Type1 gives the type of mnemonic symbol. Type1 can be one of the followings :

   IS – 1
   DL – 2
   AD – 3

2. **Code1 :** The field Code1 gives the index of the mnemonic symbol in the MOT.

Similarly, Type2 gives the type of the second symbol and Code2 gives the reference of the corresponding table. A constant is stored as it is in IC. Type3 and Code3 have the similar interpretation.

Algorithm :

1. LC = 0 ; // location counter
2. iPT = 0 ; // index of next entry in pool table
3. PT [iPT] = 0 ; // pool 0 starts from location 0 // in the literal table
4. iLT = 0 ; // index of next entry in the literal table
5. Open the source file

```
printf("\n enter the source file :") ;
gets (file1) ;
ptr1 = fopen (file1, "r") ;
```

6. while (!end of the source file)
   {
   a) Read the next line and store the same in a character array 'nextline' after substituting special characters with blanks.

```
i = 0 ;
nextline[i] = fgetc(ptr1);
while( nextline[i] != ' \n' && nextline [i] != EOF )
{
    if ( !isalnum(nextline[i]) )
        nextline[i] = ' ';
    else
        nextline[i] = toupper(nextline[i]);
    i++ ;
    nextline[i] = fgetc(ptr1);
}
```

Scanned with CamScanner

b) if(nextline is an END statement)
break;

```
sscanf(nextline, " %s", SI);
if(strcmp(SI,"END") == 0)
    break;
```

c) if the nextline contains a label then it is stored in the symbol table (ST).

```
if (searchMOT (SI) == -1)
    insert (SI, LC, 0);
```

d) Separate opcode and operands.

```
if (label is present)
    sscanf(nextline, "%s%s%s%s", label, S1, S2, S3);
else
    sscanf(nextline, "%s%s%s", S1, S2, S3);
```

e) If the statement stored in nextline is a declaration statement then

(i) Type1 = type of the declaration statement

(ii) Code1 = code (index in MOT) of the declaration statement.

(iii) Size = size of the memory area required by DS/DC.

(iv) Generate intermediate code and store the same in IC table.

(v) LC = LC + size.

```
void DC( )
{
  int index :
  index = searchMOT (s1) ;
  IC[iIC].Type1 = IC[iIC]. Type2 = IC[iIC]. Type3 = 0;
//initialize
  IC[iIC].LC = LC;
  IC[iIC].Code1 = index;
  IC[iIC].Type1 = MOT[index]. Class ;
  IC[iIC].Type2 = 6;    //6 IS TYPE FOR CONSTANTS
  IC[iIC].Code2 = atoi(s2) ;
  index = searchST(label) ;
  if (index == - 1)
    index = insertST(label, 0, 0) ;
  ST[index]. Address = LC ;
  ST[index]. Size = 1 ;
  LC = LC + 1 ;
  iTC++ ;
}
void DS()
{
  int index ;
  index = searchMOT(s1) ;
  IC[iIC].Type1 = IC[iIC]. Type2 = IC[iIC]. Type3 = 0;
  //initialize
  IC[iIC].LC = LC;
  IC[iIC].Code1 = index ;
  IC[iIC].Type1 = MOT[index]. Class ;
  IC[iIC].Type2 = 6;    //6 IS TYLPE FOR CONSTANTS
  IC[iIC].Code2 = atoi (s2);
  index = searchST (label);
  if(index == - 1)
    index = insertST(label, 0, 0);
  ST[index].Address = LC ;
  ST[index].Size = atoi(s2) ;
  LC = LC+atoi (s2) ;
  iIC++ ;
}
```

f) If the statement stored in nextline is an imperative statement then

(i) Type1 = type of the imperative statement

(ii) Code1 = code(index in MOT) of the opcode

(iii) Size = size of the instruction

(iv) If operand is a literal then the literal is stored in the literal table

(v) Generate intermediate code and store the same in the IC table

(vi) LC = LC + size

```
void imperative()
{
  int index ;
  index = searchMOT(s1);
  IC[iIC].Type1 = IC[iIC].Type2 = IC[iIC].Type3 = 0;
//initialize
  IC[iIC].LC = LC ;
  IC[iIC].Code1 = index ;
  IC[iIC].Type1 = MOT[index].Class ;
  LC = LC+1 ;
  if(tokencount >1)
    {
      index = searchMOT (s2) ;
      if(index != - 1)
      {
          IC[iIC].Code2 = index ;
          IC[iIC].Type2 = MOT[index].Class ;
      }
```

Scanned with CamScanner

```
        else
            {   //It is a variable
                index = searchST(s2) ;
                if(index == – 1)
                   index = insertST(s2, 0, 0) ;
                IC[iIC].Code2 = index ;
                IC[iIC].Type2 = 7;
  //VALUE 7 IS FOR VARIABLES
            }
        }
  if(tokencount > 2)
  {
    if(isdigit(*s3))
      {
          LT[iLT].Literal = atoi(s3) ;
          IC[iIC].Code3 = iLT ;
          IC[iIC].Type3 = 8 ;
  //VALUE 8 IS FOR LITERAL TYPE
          iLT++ ;
      }
      else
      {
          index = searchST (s3) ;
          if(index == – 1)
              index = insertST(s3, 0, 0) ;
          IC[iIC].Code3 = index;
          IC[iIC].Type3 = 7 ;
  //VALUE 7 IS FOR VARIABLES
      }
  }
  iIC++ ;
}
```

g) If the statement stored in nextline is an LTORG statement then

(i) Literals of the current pool of literals are assigned addresses

(ii) iPT = iPT + 1

(iii) PT[iPT] = iLT

(iv) LC is modified suitably.

```
void LTORG()
{
  int i, index;
  for(i = PT[iPT]; i < iLT ; i++)
  {
    LT[i]. Address = LC ;
    index = searchMOT ("DC" );
  IC[iIC].Type1 = IC[iIC].Type2 = IC[iIC].Type3 = 0;
  //initialize
    IC[iIC].LC = LC ;
    IC[iIC].Code1 = index ;
    IC[iIC].Type1 = MOT[index]. Class ;
    IC[iIC].Type2 = 6 ; //6 IS TYPE FOR CONSTANTS
    IC[iIC].Code2 = LT[i].Literal ;
    LC = LC+1 ;
    iIC++ ;
    }
  iPT++ ;
  PT[iPT] = iLT ;
}
```

h) If the statement stored in the nextline is a START or ORIGIN statement then

(i) LC = address specified in the statement

(ii) Generate intermediate code and store the same in the IC table.

```
void START()
{
  int index ;
  index = searchMOT(s1) ;
  IC[iIC].Type1 = IC[iIC].Type2 = IC[iIC]. Type3 = 0;
  //initialize
  IC[iIC].LC = LC ;
  IC[iIC].Code1 = index ;
  IC[iIC].Type1 = MOT[index].Class ;
  IC[iIC].Type2 = 6 ;   // 6 IS TYPE FOR CONSTANTS
  IC[iIC].Code2 = atoi (s2) ;
  LC = atoi(s2) ;
  iIC++ ;
}
```

i) If the statement stored in nextline is an EQU statement then

(i) Address of the variable is corrected in the symbol table. New address is extracted from the statement.

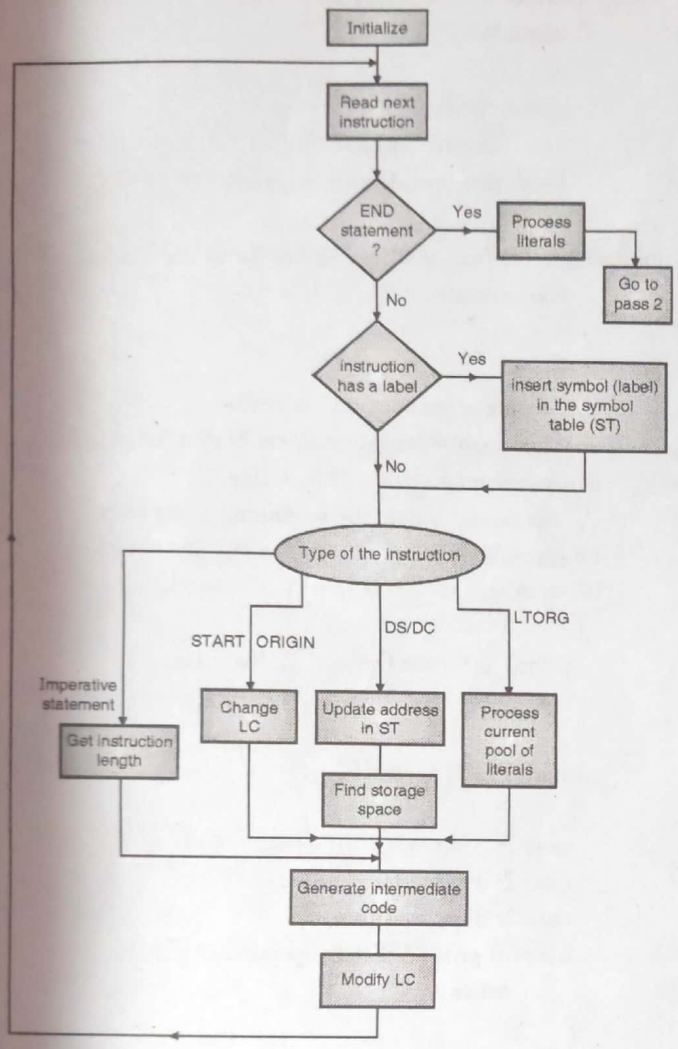
(ii) Generate intermediate code and store the same in the IC table.

}

Scanned with CamScanner

7. Processing of END statement
   (i) Perform step 6(g)
   (ii) Go to pass II

2.4.3 Flowchart for Pass I



(S2.12)**Fig. 2.4.7**

2.4.4 C-Program for Pass I

➢ **Program 2.4.1 : Program for pass I**

```
#include<stdio.h>
#include<string.h>
struct MOTtable
 {
   char Mnemonic[6];
   int  Class;
   char Opcode[3];
 };

struct symboltable
 {
   char Symbol[8];
   int  Address;
   int  Size;
 }ST[20];
struct literaltable
 {
   int  Literal;
   int  Address;
 }LT[10];
int PT[20];
struct intermediatecode
 {
  int LC;
  int Code1,Type1;
  int Code2,Type2;
  int Code3,Type3;
 }IC[30];


static struct MOTtable
MOT[28]={{"STOP",1,"00"},{"ADD",1,"01"},{"SUB",1,"0
2"},
{"MULT",1,"03"},{"MOVER",1,"04"},{"MOVEM",1,"05"},
   {"COMP",1,"06"},{"BC",1,"07"},{"DIV",1,"08"},
   {"READ",1,"09"},{"PRINT",1,"10"},
   {"START",3,"01"},{"END",3,"02"},{"ORIGIN",3,"03"},
   {"EQU",3,"04"},{"LTORG",3,"05"},
   {"DS",2,"01"},{"DC",2,"02"},
   {"AREG",4,"01"},{"BREG",4,"02"},{"CREG",4,"03"},

"EQ",5,"01"},{"LT",5,"02"},{"GT",5,"03"},{"LE",5,"04"},
   {"GE",5,"05"},{"NE",5,"06"},{"ANY",5,"07"}};
int nMOT=28; //Number of entries in MOT
int LC=0;   //Location counter
int iPT;    //Index of next entry in Pool Table
int iLT=0;  //Index of next entry in Literal Table
int iST=0;  //Index of next entry in Symbol Table
int iIC=0;  //Index of next entry in intermediate code table
int searchST(char symbol[])
 {
   int i;
   for(i=0;i<iST;i++)
     if(strcmp(ST[i].Symbol,symbol)==0)
        return(i);
   return(-1);
 }

int searchMOT(char symbol[])
 {
   int i;
   for(i=0;i<nMOT;i++)
    if(strcmp(MOT[i].Mnemonic,symbol)==0)
      return(i);
```

Scanned with CamScanner

```
    return(-1);
  }
int insertST( char symbol[],int address,int size)
  {
    strcpy(ST[iST].Symbol,symbol);
    ST[iST].Address=address;
    ST[iST].Size=size;
    iST++;
    return(iST-1);
  }

void imperative();          //Handle an executable statement
void declaration();         //Handle a declaration statement
void directive();       //Handle an assembler directive
void print_symbol(); //Display symbol table
void print_pool();     //Display pool table
void print_literal();       //Display literal table
void print_opcode(); //Display opcode table
void intermediate(); //Display intermediate code
char s1[8],s2[8],s3[8],label[8];
void LTORG();               //Handle LTORG directive
void DC();              //Handle declaration statement DC
void DS();              //Handle declaration statement DS
void START();               //Handle START directive
int tokencount;         //total number of words in a statement
void main()
{
   char file1[40],nextline[80];
   int len,i,j,temp,errortype;
   FILE *ptr1;
   clrscr();
   printf("\nenter Source file name:");
   gets(file1);
   ptr1=fopen(file1,"r");
   while(!feof(ptr1))
    {
   /*Read a line of assembly program and remove special
    characters*/
     i=0;
     nextline[i]=fgetc(ptr1);
     while(nextline[i]!='\n'&& nextline[i]!=EOF )
       {
            if(!isalnum(nextline[i]))
                nextline[i]=' ';
            else
                nextline[i]=toupper(nextline[i]);
            i++;
            nextline[i]=fgetc(ptr1);
       }
     nextline[i]='\0';

//if the nextline is an END statement
     sscanf(nextline,"%s",s1);
                          //read from the nextline in s1
     if(strcmp(s1,"END")==0)
        break;
//if the nextline contains a label
     if(searchMOT(s1)==-1)
       {
       if(searchST(s1)==-1)
            insertST(s1,LC,0);
       //separate opcode and operands

tokencount=sscanf(nextline,"%s%s%s%s",label,s1,s2,s3);
        tokencount--;
       }
     else
       //separate opcode and operands
       tokencount=sscanf(nextline,"%s%s%s",s1,s2,s3);
     if(tokencount==0)    //blank line
        continue;  //goto the beginning of the loop
     i=searchMOT(s1);
     if(i == -1)
      {
       printf("\nWrong Opcode .... %s",s1);
       continue;
      }
     switch(MOT[i].Class)
       {
       case 1:   imperative();break;
       case 2: declaration();break;
       case 3: directive();break;
       default: printf("\Wrong opcode ...%s",s1);
            break;
       }

   }
 LTORG();
 print_symbol();
 getch();
 print_pool();
 getch();
 print_literal();
 getch();
 print_opcode();
 getch();
 intermediate();
 getch();
}

void imperative()
 {
  int index;
  index=searchMOT(s1);
```

Scanned with CamScanner

```
IC[iIC].Type1=IC[iIC].Type2=IC[iIC].Type3=0;
                                        //intialize
IC[iIC].LC=LC;
IC[iIC].Code1=index;
IC[iIC].Type1=MOT[index].Class;
LC=LC+1;
if(tokencount>1)
 {
  index=searchMOT(s2);
  if(index != -1)
  {
    IC[iIC].Code2=index;
    IC[iIC].Type2=MOT[index].Class;
  }
 else
  {  //It is a variable
    index=searchST(s2);
    if(index==-1)
     index=insertST(s2,0,0);
   IC[iIC].Code2=index;
   IC[iIC].Type2=7; //VALUE 7 IS FOR VARIABLES
  }
 }
if(tokencount>2)
 {
  if(isdigit(*s3))
    {
   LT[iLT].Literal=atoi(s3);
   IC[iIC].Code3=iLT;
   IC[iIC].Type3=8;
          //VALUE 8 IS FOR LITERAL TYPE
   iLT++;
    }
  else
    {
     index=searchST(s3);
     if(index==-1)
       index=insertST(s3,0,0);
     IC[iIC].Code3=index;
     IC[iIC].Type3=7;
          //VALUE 7 IS FOR VARIABLES
    }
  }

 iIC++;
}

void declaration()
{
 if(strcmp(s1,"DC")==0)
  {
   DC();
    return;
   }
  if(strcmp(s1,"DS")==0)
    DS();
}

void directive()
{
  if(strcmp(s1,"START")==0)
    {
     START();
     return;
    }
  if(strcmp(s1,"LTORG")==0)
    LTORG();

}

void LTORG()
{
  int i,index;
  for(i=PT[iPT];i<iLT;i++)
   {
     LT[i].Address=LC;
     index=searchMOT("DC");
     IC[iIC].Type1=IC[iIC].Type2=IC[iIC].Type3=0;
   //intialize
     IC[iIC].LC=LC;
     IC[iIC].Code1=index;
     IC[iIC].Type1=MOT[index].Class;
     IC[iIC].Type2=6;   //6 IS TYPE FOR CONSTANTS
     IC[iIC].Code2=LT[i].Literal;
     LC=LC+1;
     iIC++;
   }
  iPT++;
  PT[iPT]=iLT;

}
void DC()
{
   int index;
   index=searchMOT(s1);
   IC[iIC].Type1=IC[iIC].Type2=IC[iIC].Type3=0;
   //intialize
   IC[iIC].LC=LC;
   IC[iIC].Code1=index;
   IC[iIC].Type1=MOT[index].Class;
   IC[iIC].Type2=6;       //6 IS TYPE FOR CONSTANTS
   IC[iIC].Code2=atoi(s2);
   index=searchST(label);
   if(index==-1)
```

Scanned with CamScanner

```
    index=insertST(label,0,0);
  ST[index].Address=LC;
  ST[index].Size=1;
  LC=LC+1;
  iIC++;
}

void DS()
{
  int index;
  index=searchMOT(s1);
  IC[iIC].Type1=IC[iIC].Type2=IC[iIC].Type3=0;
  //intialize
  IC[iIC].LC=LC;
  IC[iIC].Code1=index;
  IC[iIC].Type1=MOT[index].Class;
  IC[iIC].Type2=6;      //6 IS TYPE FOR CONSTANTS
  IC[iIC].Code2=atoi(s2);
  index=searchST(label);
  if(index==-1)
    index=insertST(label,0,0);
  ST[index].Address=LC;
  ST[index].Size=atoi(s2);
  LC=LC+atoi(s2);
  iIC++;
}


void START()
{
 int index;
 index=searchMOT(s1);
 IC[iIC].Type1=IC[iIC].Type2=IC[iIC].Type3=0;
//intialize
 IC[iIC].LC=LC;
 IC[iIC].Code1=index;
 IC[iIC].Type1=MOT[index].Class;
 IC[iIC].Type2=6;      //6 IS TYPE FOR CONSTANTS
 IC[iIC].Code2=atoi(s2);
 LC=atoi(s2);
 iIC++;
}

void intermediate()
{  int i;
   char decode[9][3]=
        {" ","IS","DL","AD","RG","CC","C","S","L"};
   printf("\n\Intermediate Code :");
   for(i=0;i<iIC;i++)
    {

   printf("\n%3d)(%s,%2s)",IC[i].LC,decode[IC[i].Type1]
                ,MOT[IC[i].Code1].Opcode);
      if(IC[i].Type2!=0)
       {
         if(IC[i].Type2<6)
            printf(" (%s,%2s)",decode[IC[i].Type2]
                  ,MOT[IC[i].Code2].Opcode);

         else
            printf("  (%s,%2d)",decode[IC[i].Type2]
                  ,IC[i].Code2);

       }
      if(IC[i].Type3!=0)
            printf("  (%s,%2d)",decode[IC[i].Type3]
                  ,IC[i].Code3);

     }
}
void print_symbol()
 {
    int i;
    printf("\*******symbol table ***********\n");
    for(i=0;i<iST;i++)
      printf("\n%10s  %3d
        %3d",ST[i].Symbol,ST[i].Address,ST[i].Size);
}
void print_pool()
 {
    int i;
    printf("\npool table *********\n");
    for(i=0;i<iPT;i++)
      printf("\n%d",PT[i]);
 }
 void print_literal()
 {
    int i;
    printf("\nliteral table*******\n");
    for(i=0;i<iLT;i++)
      printf("\n%5d\t%5d",LT[i].Literal,LT[i].Address);
 }
void print_opcode()
 {
    int i;
    printf("\nopcode table ************");
    for(i=0;i<nMOT;i++)
      if(MOT[i].Class==1)
    printf("\n%6s 2s",MOT[i].Mnemonic,MOT[i].Opcode);

 }
```

Scanned with CamScanner

Input / Output

```
enter Source file name:xx.asm

*******symbol table **********

     L1  100   0
      X  108   1
      Y  109   2

pool table *********

0
2

literal table*******

   5    103
   2    104
   4    111

opcode table *************
 STOP  00
  ADD  01
  SUB  02
 MULT  03
MOVER  04
MOVEM  05
 COMP  06
   BC  07
  DIV  08
 READ  09
PRINT  10

Intermediate Code :
  0)  (AD,01)  (C,100)
100)  (IS,04) (RG,01)  (L, 0)
101)  (IS,05) (RG,02)  (S, 1)
102)  (IS,02) (RG,01)  (L, 1)
103)  (DL,02)  (C, 5)
104)  (DL,02)  (C, 2)
105)  (IS,04) (RG,01)  (S, 2)
106)  (IS,07) (CC,07)  (S, 0)
107)  (IS,01) (RG,03)  (L, 2)
108)  (DL,02)  (C, 5)
109)  (DL,01)  (C, 2)
111)  (DL,02)  (C, 4)
```

```
Contents of the file xx.asm
---------------------------
START 100
L1  MOVER AREG,=5
    MOVEM BREG X
    SUB   AREG,=2
    LTORG
    MOVER AREG Y
    BC any,L1
    ADD CREG,4
X   DC 5
Y   DS 2
    END
```

2.4.5 Pass II of the Assembler

➔ **(May 2014)**

Q. Design and explain assembler pass-II of two pass assembler in detail. **SPPU - May 2014, 8 Marks**

Algorithm for pass II assumes that the intermediate code is stored in the table IC[ ]. Target code will be assembled in the area named code-area.

☞ **Algorithm**

1. Code_area_address = address of Code_area
2. For each entry in IC[ ]
   {
   a) If an imperative statement
      (i) Read LC
      (ii) Get opcode
      (iii) Get operand / literal address from the symbol / literal table.
      (iv) Assemble instruct in machine code_buffer.
      (v) Move contents of machine_code_buffer in code_area at the address LC + code_area_address.
   b) If a DC statement then
      (i) Read LC
      (ii) Assemble the constant in machine_code_buffer.
      (iii) Move contents of machine_code_buffer in code_area at the address LC+ code_area_address.
3. Write code_area into output file.

2.4.6 C-Program for Pass II

➢ **Program 2.4.2 : Program for pass II**

```
#include<stdio.h>
#include<string.h>
struct MOTtable
 {
```

Scanned with CamScanner

```
    char Mnemonic[6];
    int  Class;
    char Opcode[3];
};


struct symboltable
 {
    char Symbol[8];
    int  Address;
    int  Size;
 }ST[20];
struct literaltable
 {
    int  Literal:
    int  Address;
 }LT[10];
int PT[20];
struct intermediatecode
 {
   int LC;
   int Code1,Type1;
   int Code2,Type2;
   int Code3,Type3;
 }IC[30];
int LC=0;   //Location counter
int iPT;    //Index of next entry in Poll Table
int iLT=0;  //Index of next entry in Literal Table
int iST=0;  //Index of next entry in Symbol Table
int iIC=0;  //Index of next entry in intemediate code table

void main()
 {
    char file1[40],nextline[80];
    char code1 [5],code2 [5],code3 [5],code4 [5],
    code5 [5],code6 [5];
    int len,i,j,temp,errortype,count;
    FILE *ptr1;
    printf("\nEnter Symbol Table Entries : ");
    printf("\nEnter No. entries : ");
    scanf("%d",&count);
    iST=count;
    for(i=0;i<count;i++)
     {
      printf("\nEnter the address of the symbol : ");
      scanf("%d",&(ST[i].Address));
     }
    printf("\nEnter Literal Table Entries : ");
    printf("\nEnter No. entries : ");
    scanf("%d",&count);
    iLT=count;
    for(i=0;i<count;i++)
     {
      printf("\nEnter the address of the literal : ");
      scanf("%d",&(LT[i].Address));
     }

    printf("\nenter Source file name(containing Intermediate
Code):");
    flushall();
    gets(file1);
    ptr1=fopen(file1,"r");
    while(!feof(ptr1))
     {
/* Read a line of intermediate code and remove special */
      characters
      i=0;
      nextline[i]=fgetc(ptr1);
      while(nextline[i]!='\n'&& nextline[i]!=EOF )
        {
            if(!isalnum(nextline[i]))
                nextline[i]=' ';
            else
                nextline[i]=toupper(nextline[i]);
            i++;
            nextline[i]=fgetc(ptr1);
        }
      nextline[i]='\0';


    count=sscanf(nextline,"%s%s%s%s%s%s",code1,code
2,code3,code4,code5,code6);
      if(strcmp(code1,"AD")==0 &&
strcmp(code2,"01")==0)
        {
        LC=atoi(code4);
        continue;
        }

      if(strcmp(code1,"IS")==0 )
        {
        printf("\n%3d)  %s ",LC,code2);
        if(count==2)
         {
            printf("00 000");
            LC=LC+1;
            continue;
         }
        if(count==4)
         {
           strcpy(code5,code3);
           strcpy(code6,code4);
```

Scanned with CamScanner

```
            printf("00 ");
          }
        else
            printf("%s ",code4);
        if(strcmp(code5,"S")==0)
          printf("%d",ST[atoi(code6)].Address);
        else
          printf("%d",LT[atoi(code6)].Address);

        LC=LC+1;
        continue;
         }
      if(strcmp(code1,"DL")==0 )
        {
        printf("\n%3d)  ",LC);
        if(strcmp(code2,"01")==0)
            LC=LC+atoi(code4);
        else
         {
            printf("00 00 %3s",code4);
            LC=LC+1;
         }
        }

    }
}
```

Input / Output

```
Enter Symbol Table Entries :
Enter No. entries : 4

Enter the address of the symbol : 101

Enter the address of the symbol : 132

Enter the address of the symbol : 103

Enter the address of the symbol : 112

Enter Literal Table Entries :
Enter No. entries : 1

Enter the address of the literal : 108

enter Source file name(containing Intermediate Code):ic.ic


100)  09 00 101
101)  04 02 108
102)  05 02 132
103)  03 02 132
104)  04 03 132
105)  06 03 101
106)  07 04 103
107)  05 02 112
108)  00 00   1
109)  10 00 112
110)  00 00 000
111)
112)
132)

Contents of the file ic.ic
--------------------------
(AD,01) (C,100)
(IS,09) (S,0)
(IS,04) (RG,02) (L,0)
(IS,05) (RG,02) (S,1)
(IS,03) (RG,02) (S,1)
(IS,04) (RG,03) (S,1)
(IS,06) (RG,03) (S,0)
(IS,07) (CC,04) (S,2)
(IS,05) (RG,02) (S,3)
(DL,02) (C,1)
(IS,10) (S,3)
(IS,00)
(DL,01)(C,1)
(DL,01) (C,20)
(DL,01) (C,1)
(AD,02)
```

Example 2.4.1

For the following assembly language code show the contents of symbol table, literal table and also generate intermediate and target code. [Assume suitable op-codes and instruction length and clearly indicate the assumptions made]

```
          START   1000
          READ    N
          MOVER   B, ="1"
          MOVEM   B, TERM
AGAIN     MUL     B, TERM
          MOVER   C, TERM
          COMP    C, N
          BC      LE, AGAIN
          MOVEM   B, RESULT
          LTORG
          PRINT   RESULT
          STOP
N         DS      1
RESULT    DS      20
TERM      DS      1
          END
```

Scanned with CamScanner

Solution :

Step 1 :

		Assembly program		LC	Intermediate Code
1.		START	1000		(AD, 01) (C, 1000)
2.		READ	N	1000	(IS, 09) (S, 0)
3.		MOVER	B, = "1"	1001	(IS, 04) (RG, 02) (L, 0)
4.		MOVEM	B, TERM	1002	(IS, 05) (RG, 02) (S, 1)
5.	AGAIN	MUL	B, TERM	1003	(IS, 03) (RG, 02) (S, 1)
6.		MOVER	C, TERM	1004	(IS, 04) (RG, 03) (S, 1)
7.		COMP	C, N	1005	(IS, 06) (RG, 03) (S, 0)
8.		BC	LE, AGAIN	1006	(IS, 07) (CC, 04) (S, 2)
9.		MOVEM	B, RESULT	1007	(IS, 05) (RG, 02) (S, 3)
10.		LTORG		1008	(DL, 02) (C, 1)
11.		PRINT	RESULT	1009	(IS, 10) (S, 3)
12.		STOP		1010	(IS, 00)
13.	N	DS	1	1011	(DL, 01) (C, 1)
14.	RESULT	DS	20	1012	(DL, 01) (C, 20)
15.	TERM	DS	1	1032	(DL, 01) (C, 1)
16.		END		1033	(AD, 02)

Symbol Table

	Symbol	Address
0	N	1011
1	TERM	1032
2	AGAIN	1003
3	RESULT	1012

Literal Table

	Literal	Address
0	= '1'	1008

Pool Table

0

Step 2 :

	Intermediate code	LC	Machine code		
1.	(AD, 01) (C, 1000)				
2.	(IS, 09) (S, 0)	1000	09	00	1011
3.	(IS, 04) (RG, 02) (L, 0)	1001	04	02	1008
4.	(IS, 05) (RG, 02) (S, 1)	1002	05	02	1032
5.	(IS, 03) (RG, 02) (S, 1)	1003	03	02	1032
6.	(IS, 04) (RG, 03) (S, 1)	1004	04	03	1032
7.	(IS, 06) (RG, 03) (S, 0)	1005	06	03	1011
8.	(IS, 07) (CC, 04) (S, 2)	1006	07	04	1003
9.	(IS, 05) (RG, 02) (S, 3)	1007	05	02	1012
10.	(DL, 02) (C, 1)	1008	00	00	0001
11.	(IS, 10) (S, 3)	1009	10	00	1012
12.	(IS, 00)	1010	00	00	0000
13.	(DL, 01) (C, 1)	1011			
14.	(DL, 01) (C, 20)	1012			
15.	(DL, 01) (C, 1)	1032			
16.	(AD, 02)				

Example 2.4.2

	START	100
	MOVER	AREG, = 5
	ADD	CREG, = 1
A	DS	3
L1	MOVER	AREG, B
	ADD	AREG, C
	MOVEM	AREG, D
	LTORG	
D	EQU	A + 1
L2	PRINT	D
	ORIGIN	A – 1
	SUB	AREG, = 1
	MULT	CREG, B
C	DC	'5'
	ORIGIN	L2 + 1
	STOP	
B	DC	19
	END	

(i) Show the contents of symbol table, literal table and pool table at the end of pass I.

(ii) Show the intermediate code generated for the program.

(iii) Show the machine code generated for the program

Solution :

Step 1 :

		Assembly code		LC	Intermediate code
1.		START	100		(AD, 01) (C, 100)
2.		MOVER	AREG, = 5	100	(IS, 04) (RG, 01) (L, 0)
3.		ADD	CREG, = 1	101	(IS, 01) (RG, 03) (L, 1)
4.	A	DS	3	102	(DL, 01) (C, 3)
5.	L1	MOVER	AREG, B	105	(IS, 04) (RG, 01) (S, 2)
6.		ADD	AREG, C	106	(IS, 01) (RG, 01) (S, 3)
7.		MOVEM	AREG, D	107	(IS, 05) (RG, 01) (S, 4)
8.		LTORG		108	(DL, 02) (C, 5)
				109	(DL, 02) (C, 1)
9.	D	EQU	A + 1		(AD, 04) (C, 103)
10.	L2	PRINT	D	110	(IS, 10) (S, 4)
11.		ORIGIN	A – 1		(AD, 03) (C, 101)
12		SUB	AREG, = 1	101	(IS, 02) (RG, 01) (L, 2)
13.		MULT	CREG, B	102	(IS, 03) (RG, 03) (S, 2)
14.	C	DC	'5'	103	(DL, 02) (C, 5)
15.		ORIGIN	L2 + 1		(AD, 03) (C, 111)
16.		STOP		111	(IS, 00)
17.	B	DC	19	112	(DL, 02) (C, 19)
18.		END			(AD, 02)
19.		LTORG		113	(DL, 02) (C, 1)

Scanned with CamScanner

	Symbol Table	
	Symbol	Address
0	A	102
1	L1	105
2	B	112
3	C	103
4	D	103

	Literal Table	
	Literal	Address
0	= 5	108
1	= 1	109
2	= 1	113

	Pool Table
0	0
1	2

Step 2 :

	Intermediate code	LC	Machine code		
1.	(AD, 01) (C, 100)				
2.	(IS, 04) (RG, 01) (L, 0)	100	04	01	108
3.	(IS, 01) (RG, 03) (L, 1)	101	01	03	109
4.	(DL, 01) (C, 3)	102			
5.	(IS, 04) (RG, 01) (S, 2)	105	04	01	112
6.	(IS, 01) (RG, 01) (S, 3)	106	01	01	103
7.	(IS, 05) (RG, 01) (S, 4)	107	05	01	103
8.	(DL, 02) (C, 5)	108	00	00	005
	(DL, 02) (C, 1)	109	00	00	001
9.	(AD, 04) (C, 103)				
10.	(IS, 10) (S, 4)	110	10	00	103
11.	(AD, 03) (C, 101)				
12.	(IS, 02) (RG, 01) (L, 2)	101	02	01	113
13.	(IS, 03) (RG, 03) (S, 2)	102	03	03	112
14.	(DL, 02) (C, 5)	103	00	00	005
15.	(AD, 03) (C, 111)				
16.	(IS, 00)	111	00	00	000
17.	(DL, 02) (C, 19)	112	00	00	019
18.	(AD, 02)				
19.	(DL, 02) (C, 1)	113	00	00	001

2.4.7 Error Reporting

An assembly program may contain errors. It may be necessary to report these errors effectively. Some errors can be reported during processing in pass I. Some errors can only be reported at the end of the source program. Some of the typical errors include :

1. Syntax errors like missing commas etc.
2. Invalid opcode.
3. Duplicate definition of a symbol.
4. Undefined symbol.
5. Missing START statement.
6. Missing END statement.
7. Symbol defined but not used.

- Errors like undefined symbol, missing START statement and missing END statement can only be reported at the end of the source program.
- A sample program with error reporting is shown in Fig. 2.4.8.

Sr. No.		Statement	Address
001		START 100	
002		MOVER AREG, X	
003		ADDER BREG,X	
		***error *** invalid opcode	
004		ADD AREG, Y	
005	X	DC '2'	
006	X	DS 1	
		error duplicate definition of X	
007	Z	DC '3'	
		END	
	***error *** undefined symbol Y		
	***error *** symbol Z is not used.		

Fig. 2.4.8 : Error reporting in pass I

Example 2.4.3

With respect to the design of a two pass assembler, state whether the following statements are true or false. Justify your answer in each case.

i) Literals are processed in PASS-II.

ii) Undefined symbols are detected in PASS-I.

iii) Incorrect op-codes are identified in PASS-I.

Solution :

i) Literals are processed in Pass-II

– No.

Literals are processed in Pass-I and not in Pass-II. Literals are stored in literal table in pass-I. These literals are accommodated in memory on processing of LTORG directive.

ii) Undefined symbols are detected in Pass-I

– Yes.

At the end of Pass-I, undefined symbols can be detected. Undefined symbols can be detected by introducing one more column in the symbol table. Value of this column-entry can be set to true if the corresponding symbol is defined using DC/DS.

iii) Incorrect op-codes are identified in Pass-I

– True.

Processing of opcode is carried out in Pass-I.]

Example 2.4.4

State true or false.

(i) The output of an assembler is an object program.

(ii) The EQU statement defines the symbol to represent the constant in the operand specification.

Solution :

(i) True

(ii) True.

Scanned with CamScanner

Example 2.4.5

Is it necessary to have an explicit representation of DS statements and assembler directives in intermediate code ? Explain.

Solution :

If the intermediate code contains the address field (LC value) then the representation of DS statements in intermediate code is not necessary. With the address field of intermediate code omitted, a representation for the DS statements and assembler directives become necessary. It will be needed for LC processing in Pass-II.

2.5 Exam Pack (University Questions)

☞ **Syllabus Topic : Elements of Assembly Language Programming**

Q. Explain in brief imperative statements, declaration statements and assembler directives with examples for assembly language programming. *(Refer section 2.1.3)* (9 Marks) **(May 2017)**

Q. Explain different assembly language statements with examples. *(Refer section 2.1.3)* (7 Marks) **(Dec. 2016)**

Q. Explain in brief assembler directives with examples. *(Refer section 2.1.3)* (6 Marks) **(May 2016)**

Q. What is an assembler ? *(Refer section 2.2)* (4 Marks) **(Dec. 2013)**

☞ **Syllabus Topics : Simple Assembler Scheme, Structure of an Assembler**

Q. Explain ORIGIN statement with example. *(Refer section 2.3(1))* (2 Marks) **(May 2014, Aug. 2015(In Sem), Oct. 2016(In Sem))**

Q. Explain EQU statement. *(Refer section 2.3(2))* (2 Marks) **(May 2014, Aug. 2015(In Sem))**

Q. What is a LTORG Statement ? Explain with example. *(Refer section 2.3(3))* (8 Marks) **(May 2014, Oct. 2016(In Sem))**

☞ **Syllabus Topic : Design of Two Pass Assembler**

Q. How Pass I of an assembler works. *(Refer section 2.4)* (4 Marks) **(May 2013)**

Q. Explain two pass assembler along with the schematic diagram. *(Refer section 2.4)* (4 Marks) **(May 2013)**

Q. Describe the design of Pass 1 of two pass assembler. *(Refer section 2.4)* (7 Marks) **(Dec. 2015)**

Q. Explain the algorithm in detail with data structures used. *(Refer section 2.4.2)* (4 Marks) **(May 2013)**

Q. Design and explain assembler pass-II of two pass assembler in detail. *(Refer section 2.4.5)* (8 Marks) **(May 2014)**

❑❑❑

Scanned with CamScanner

UNIT I

CHAPTER 3

Macro Processor

Syllabus Topics

Macro Definition and call, Macro expansion, Nested Macro Calls, Advanced Macro Facilities, Design of a two-pass macro-processor.

Syllabus Topic : Macro Definition

3.1 Macro Definition

→ (May 2014)

Q. Explain the term macro definition.

SPPU - May 2014, 2 Marks

Macro allows a sequence of source language code to be defined once and then referred to by name each time it is to be referred. Each time this name occurs in a program, the sequence of codes is substituted at that point.

A macro consists of :

(1) Name of the macro

(2) Set of parameters

(3) Body of macro (Code)

Parameters in a macro are optional.

For example, let us consider a program segment given below:

```
ADD AREG, X
ADD BREG, X
==========
ADD AREG, X
ADD BREG, X
==========
ADD AREG, X
ADD BREG, X
==========
```

In the above program, the sequence

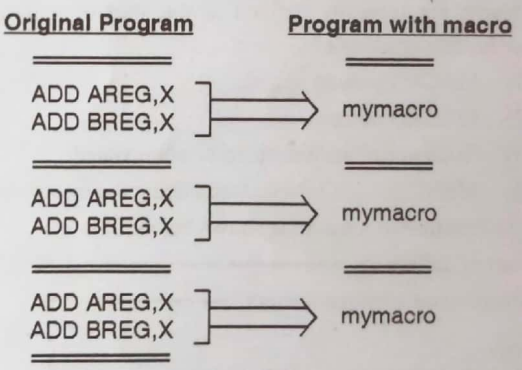
```
ADD AREG, X
ADD BREG, X
```

occurs three times. A macro allows us to attach a name to this sequence and to use this name in its place.

We can attach a name to a sequence by means of a macro definition, which is formed in the following manner :

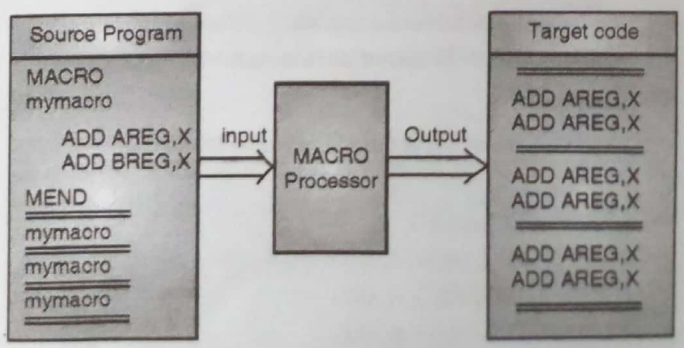
Start of definition	MACRO
macro name	mymacro
macro body	ADD AREG, X ADD BREG, X
End of macro definition	MEND

The example given above can be expressed with macro. It is given in Fig. 3.1.1



(S3.1) **Fig. 3.1.1 : Re-writing a program with macros**

A macro processor takes a source with macro definition and macro calls and replaces each macro call with its body.



(S3.2) **Fig. 3.1.2 : Macro expansion**

Scanned with CamScanner

Example 3.1.1

Compare and contrast the properties of macros and subroutines with respect to the following :

(i) Code space requirement
(ii) Execution speed
(iii) Processing required by the assembler
(iv) Flexibility and generality

Solution :

Sr. No.	Properties	Macro	Subroutines
1	Code space requirement	More, macro is expanded inline	Less, subroutines are not expanded. It is called.
2	Execution speed	More, there is no calling overhead (i.e passing of parameters and saving of return address)	Less, there is calling overhead.
3	Processing required by the assembler.	Assembler will take more time to process as code size increases	Assembler must handle passing of parameters.
4.	Flexibility and generality	A macro cannot handle labels.	A subroutine can handle every type of feature.

3.2 Defining a Macro

Macros are typically defined at the start of a program. A macro definition consists of

(1) MACRO pseudo opcode
(2) MACRO name
(3) Sequence of statements to be abbreviated
(4) MEND pseudo opcode terminating the macro definition

The structure of a macro is shown below :

Start of definition → MACRO
Macro name with parameters → [ ]
Sequence of statements to be abbreviated → { - - - - }
End of definition → MEND

- The MACRO Pseudo opcode is the first line of the macro definition.
- The next line has macro name with a list of parameters.
  < macro name > [< list of parameters>]

☞ **Example**

Fig. 3.2.1 shows the definition of macro INCR.

```
MACRO
    INCR  &ARG
    ADD   AREG, &ARG
    ADD   BREG, &ARG
    ADD   CREG, &ARG
MEND
```

Fig. 3.2.1 : A macro definition

- The macro-name line INCR &ARG indicates that :
  (1) The name of the macro is **INCR.**
  (2) There is one parameter to macro, called **ARG.**
- The parameter **ARG** does not have a default value. Such a parameter is also known as positional parameter.
- The three lines :
  ADD AREG, &ARG
  ADD BREG, &ARG
  ADD CREG, &ARG

form the body of the macro which will be used during macro expansion. During a macro call, the value of the positional parameter should be supplied.

Syllabus Topic : Macro Call

3.2.1 Calling a Macro

→ **(May 2014)**

Q. Explain the term macro calls.
SPPU - May 2014, 2 Marks

A macro is called by writing the macro name with actual parameters in an assembly program.

The macro call has the following syntax :

< macro name > [< list of parameters >]

For example,

INCR X

Will call the macro INCR with the actual parameter X.

A macro call leads to macro expansion.

Syllabus Topic : Macro Expansion

3.2.2 Macro Expansion

→ **(May 2013, May 2014)**

Q. Explain the process of Macro Expansion with relevant data structures. **SPPU - May 2013, 8 Marks**

Q. Explain the term macro expansion.
SPPU - May 2014, 2 Marks

Each call to a macro is replaced by its body.

During replacement, actual parameter is used in place of formal parameter.

- During macro expansion, each statement forming the body of the macro is picked up one by one sequentially.
- Each statement inside the macro may have :
  (1) An ordinary string, which is copied as it is during expansion.
  (2) The name of a formal parameter which is preceded by the character '&'.
- During macro expansion an ordinary string is retained without any modification. Formal parameters (string starting with &) is replaced by the actual parameter value.

Scanned with CamScanner

Consider a macro as given in Fig. 3.2.2

```
MACRO
   INCR    &VARIABLE,
&INCR_BY,&USE_REG
   MOVER     &USE_REG, &VARIABLE
   ADD       &USE_REG, &INCR_BY
   MOVEM     &USE_REG, &VARIABLE
MEND
```

Fig. 3.2.2 : A macro definition

- Name of the macro is **INCR**
- There are three positional parameters

These parameters are :

(1) VARIABLE

(2) INCR_BY

(3) USE_REG

- The body of macro **INCR** contains three statements.

Consider an assembly program with macro definition and macro-call as given in Fig. 3.2.3.

```
MACRO
    INCR &VARIABLE, &INCR_BY,&USE_REG
    MOVER     &USE_REG, &VARIABLE
    ADD       &USE_REG, &INCR_BY
    MOVEM     &USE_REG, &VARIABLE
MEND
START     100
READ      X
READ      Y
INCR      X, Y, AREG
PRINT     X
STOP
X    DS   1
Y    DS   1
END
```

Fig. 3.2.3 : An assembly program with macro

The macro processor will process the program given in Fig. 3.2.3 as explained below.

(1) The statement **START 100** will be copied as it is.

(2) The statement **READ X** will be copied as it is.

(3) The statement **READ Y** will be copied as it is.

(4) The statement **INCR X, Y, AREG** is a call to macro. The macro **INCR** will be expanded there. Values of the formal parameters are :

Formal parameter	Value
VARIABLE	X
INCR_BY	Y
USE_REG	AREG

There are three statements in the body of the macro. During expansion of the macro, actual parameters will be used instead of formal parameters.

```
MOVER   &USE_REG, &VARIABLE   expansion   MOVER   AREG, X
ADD     &USE_REG, &INCR_BY                ADD     AREG, Y
MOVEM   &USE_REG, &VARIABLE               MOVEM   AREG, X
```

(5) Remaining statements

```
PRINT X
STOP
X   DS   1
Y   DS   1
END
```

will be retained without any modification.

The output of macro processor is an expanded program with each call to macro, expanded. The output is shown in Fig. 3.2.4.

```
START     100
READ      X
READ      Y
MOVER     AREG, X   ]
ADD       AREG, Y   ]   INCR  X, Y, AREG
MOVEM     AREG, X   ]   is expanded here.

PRINT     X
STOP
X     DS    1
Y     DS    1
END
```

Fig. 3.2.4 : Expanded code

Example 3.2.1 : SPPU - Oct. 2016 (In Sem), 6 Marks

Explain lexical expansion of macro with example.

Soln. :

It is used to generate an assembly statements from a model statement. Each call to a macro is replaced by its body. During replacement, actual parameter is used in place of formal parameter.

3.2.3 Macro with Keyword Parameters

→ **(Dec. 2015)**

Q. Explain the advance macro facilities : Attributes of parameters. **SPPU - Dec. 2015, 3 Marks**

Scanned with CamScanner

A macro can have two types of parameters :

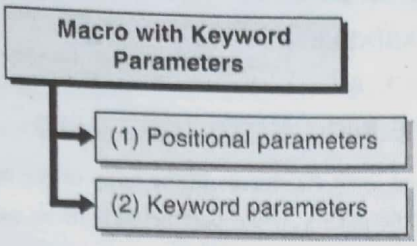


Fig. C3.1 : Keyword parameters of macros

➜ **(1) Positional parameter**

A positional parameter is written as &parameter_name. For example, in the statement

INCR &VARIABLE, &INCR_BY, &USE_REG VARIABLE, INCR_BY AND USE_REG are positional parameters.

During macro expansion, actual values of parameters are substituted on the basis their positions in the macro-call-statement.

For example, in the macro call statement

INCR X, Y, AREG

- The value X at position 1 will be assigned to the first formal parameter **VARIABLE.**
- The value Y at position 2 will be assigned to the second formal parameter **INCR_BY.**
- The value AREG at position 3 will be assigned to the third formal parameter **USE_REG.**

➜ **(2) Keyword parameter**

Keyword parameters are used for following purposes :

(1) Default value can be assigned to the parameter

(2) During a call to macro, a keyword parameter is specified by its name. It takes the following form :

< parameter name > = < parameter value >

Fig. 3.2.5 shows macro INCR of Fig. 3.2.2 using key parameters.

```
MACRO
      INCR    &VARIABLE = X,
              &INCR_BY = Y,&USE_REG = AREG
      MOVER   &USE_REG, &VARIABLE
      ADD     &USE_REG, &INCR_BY
      MOVEM   &USE_REG, &VARIABLE
MEND
```

Fig. 3.2.5 : A macro with keyword parameters

- **VARIABLE** is a keyword parameter with default value as
- **INCR_BY** is a keyword parameter with default value as Y
- **USE_REG** is a keyword parameter with default value AREG.
- The following macro calls are equivalent :

```
INCR    VARIABLE=A, INCR_BY = B,
USE_REG = BREG
INCR    INCR_BY = B, USE_REG = BREG,
        VARIABLE = A
...
INCR USE_REG = BREG, VARIABLE = A, INCR_BY = B
```

The position of keyword parameter during a macro call is n important.

- It is not necessary to pass value of every keyword parame If the value of a keyword parameter is not specified then default value is taken during expansion.

Expansion of the macro in Fig. 3.2.5 is shown under vario cases in Fig. 3.2.6.

Macro call statement	Expanded macro	
1. INCR VARIABLE=A, INCR_BY=B, USE_REG = BREG	MOVER	BREG,A
	ADD	BREG,B
	MOVEM	BREG,A
2. INCR INCR_BY=B, USE_REG=BREG, VARIABLE = A	MOVER	BREG,A
	ADD	BREG,B
	MOVEM	BREG,A
3. INCR VARIABLE = A	MOVER	AREG,A
	ADD	AREG,Y
	MOVEM	AREG,A
4. INCR	MOVER	AREG,X
	ADD	AREG,Y
	MOVEM	AREG,X

Fig. 3.2.6: Various cases of expansion

Scanned with CamScanner

3.2.4 Macro with Mixed Parameters

A macro may be defined with both :

(1) Positional Parameters
(2) Keyword Parameters

In such cases, positional parameters should be written before keyword parameters.

Fig. 3.2.7 shows the definition of macro INCR. It uses both positional and keyword parameters.

```
MACRO
    INCR      &VARIABLE, &INCR_BY,
   &USE_REG = AREG,
    MOVER     &USE_REG, &VARIABLE
    ADD       &USE_REG, &INCR_BY
    MOVEM     &USE_REG, &VARIABLE
MEND
```

Fig. 3.2.7 : A macro with mixed parameters

- The parameters **VARIABLE** and **INCR_BY** are positional parameters while **USE_REG** is a keyword parameter.
- A macro call

  INCR X, Y, USE_REG = BREG

  will assign X and Y to the positional parameters **VARIABLE** and **INCR_BY** respectively. BREG will be used as a value of the keyword parameter **USE_REG**.

3.2.5 Other Uses of Parameters

Formal parameters are not restricted to operands. Formal parameters can appear in any field of a statement inside the body of a macro. These fields include :

(1) Label (2) Opcode (3) Operand

For example, let us consider a macro definition shown in Fig. 3.2.8.

```
MACRO
    CALC      &X, &Y, &OP = ADD, &LABEL = L1
    &LABEL    MOVER     BREG, &X
              &OP BREG, &Y
              MOVEM     BREG, &X
MEND
```

Fig. 3.2.8 : A macro showing other uses of parameters

- The parameter **LABEL** appears in the label field
- The parameter **OP** appears in the opcode field
- Expansion of the macro CALC

  CALC A, B, LABEL = LOOP leads to the following lines of code :

```
LOOP  MOVER   BREG, A
      ADD     BREG, B
      MOVEM   BREG, A
```

Example 3.2.2

State true or false

(i) A macro definition consists of macro preprocessor statements.
(ii) Macro expansion is a result of a macro call.
(iii) The default flow control during macro expansion is sequential.
(iv) A macro may be defined to use both positional and keyword parameters.

Solution :

(i) True (ii) True (iii) True (iv) True

Example 3.2.3

State true or false

(i) MDT contains macro names.
(ii) Keyword parameters can be used in macro.

Solution :

(i) False, MDT need not contain macro names. Macro names are stored in MNT.
(ii) True.

Example 3.2.4

Comment on the following statements.

"Macro can not detect instructional errors i.e. errors in opcodes".

Solution : Macro pre-processor is only concerned with macro definitions and macro calls. It does not have machine opcode table.

Syllabus Topic : Nested Macro Calls

3.3 Nested Macro Calls

➔ **(Dec. 2014, May 2015, May 2016)**

Q. Explain nested macros with example.
SPPU - Dec. 2014, May 2015, May 2016, 3 Marks

A nested macro call is a macro call within a macro. There can be several levels of nesting.

- A macro containing a macro call is known as outer macro.
- A called macro is known as inner macro.
- Expansion of nested macro calls follows the LIFO (last in first out) rule.

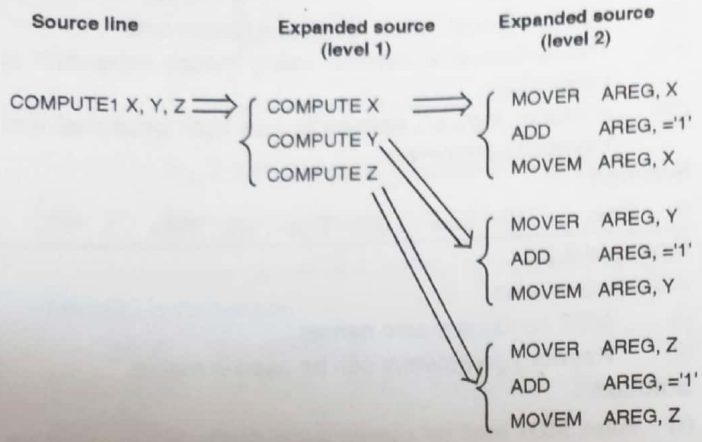
For example, let us consider the program segment shown in Fig. 3.3.1.

```
MACRO
      COMPUTE     &ARG
      MOVER       AREG, &ARG
      ADD         AREG, = '1'
      MOVEM       AREG, &ARG
MEND
MACRO
      COMPUTE1    &ARG1, &ARG2, &ARG3
      COMPUTE     &ARG1
      COMPUTE     &ARG2
      COMPUTE     &ARG3
MEND
```

Fig. 3.3.1 : Nested macros

Scanned with CamScanner

The definition of macro **COMPUTE1** contains three separate calls to a previously defined macro **COMPUTE**. Such macros are expanded on multiple levels. Expansion of COMPUTE1 X, Y, Z is shown in Fig. 3.3.2.



(S3.4)**Fig. 3.3.2 : Expansion of compute X, Y, Z**

3.3.1 Nested Macro Definition

A macro can be defined inside the body of a macro. This concept can be used for defining a group of similar macros.

- Inner macro comes into existence after a call to the outer macro.
- Inner macro can be called after it has come into existence.

A nested macro definition is shown in Fig. 3.3.3

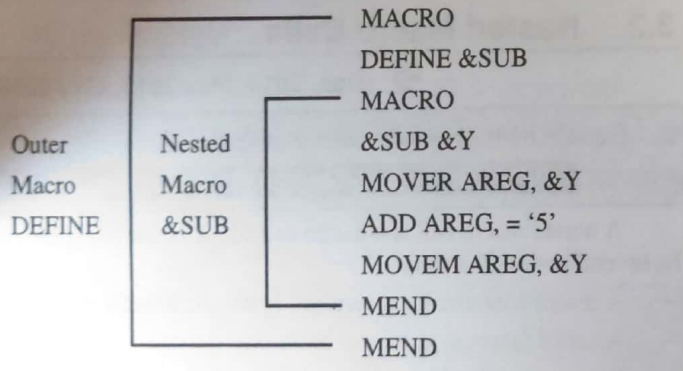


Fig. 3.3.3 : Nested macro definition

The Fig. 3.3.3 defines a macro **DEFINE,** which when called with a parameter, defines a macro with the same name as the actual parameter.

The user might call the macro with the statement

DEFINE NESTED

This will define a new macro NESTED as shown below.

```
MACRO
NESTED   &Y
MOVER    AREG, &Y
ADD      AREG, = '5'
MOVEM    AREG, &Y
MEND
```

Syllabus Topic : Advanced Macro Facilities

3.4 Advanced Macro Facilities

→ **(May 2013, Dec. 2015, May 2017)**

Q. Explain the process of alteration of flow of control during macro expansion. **SPPU - May 2013, 4 Marks**

Q. Explain the advance macro facilities : Alteration of flow of control during expansion. **SPPU - Dec. 2015, 3 Marks**

Q. Explain advanced macro facilities with example. **SPPU - May 2017, 6 Marks**

Advanced macro facilities permit conditional reordering of the sequence of macro expansion. It allows conditional selection of the machine instructions that appear in expansion of macro call.

Flow of control during macro expansion can be altered using:

(1) Conditional branch Pseudo-opcode **AIF**.

(2) Unconditional branch Pseudo- opcode **AGO**.

- AIF is similar to IF statement, the **lable** used for branching is known as **sequencing symbol.**
- A sequencing symbol has the syntax
  < ordinary string >
- AGO is similar to GO TO statement
- An AIF statement has the syntax
  AIF (< expression >) < sequencing symbol >
- An AGO statement has the syntax
  AGO < sequencing symbol >

An example showing the usage of AIF, AGO and the sequencing symbol is shown in Fig. 3.4.1.

```
MACRO
          VARY    &COUNT, &ARG1
          AIF     (&COUNT ·EQ· 1) ·ONCE
          AIF     (&COUNT ·EQ· 2) ·TWICE
          AIF     (&COUNT ·EQ· 3) ·THRICE
          AGO     ·FINAL
·ONCE     MOVER   AREG, X
    ADD   AREG, &ARG1
    AGO   ·FINAL
·TWICE    MOVER   AREG, X
    ADD   AREG, &ARG1
    ADD   AREG, &ARG1
    AGO   ·FINAL
·THRICE   MOVER   AREG, X
    ADD   AREG, &ARG1
    ADD   AREG, &ARG1
    ADD   AREG, &ARG1
·FINAL MEND
```

Fig. 3.4.1 : A macro with conditional expansion

Scanned with CamScanner

- In this macro, the number of instructions generated during expansion will depend on the value of the parameter **&count.**
- ONCE, TWICE and THRICE are sequencing symbol. They help in transfer of control during expansion of the macro.
- Various cases of macro expansions are shown below :

	Macro call	Expanded source
1.	VARY 1, Y	MOVER AREG, X ADD AREG, Y

2.	VARY 2, Y	MOVER AREG, X
		ADD AREG, Y
		ADD AREG, Y

3.	VARY 3, Y	MOVER AREG, X
		ADD AREG, Y
		ADD AREG, Y
		ADD AREG, Y

- AIF and AGO statements do not appear in the expanded source. AIF and AGO statements control the sequence in which the macro processor expands the statements during expansion.
- Sequencing symbols do not appear in the expanded code.

3.4.1 Expansion Time Variables (EV)

➔ **(Dec. 2014, May 2015, Dec. 2015)**

Q. Explain expansion time variables with example.
SPPU - Dec. 2014, May 2015, 3 Marks

Q. Explain the advance macro facilities : Expansion time variables **SPPU - Dec. 2015, 3 Marks**

Expansion time variables are used during macro expansions. These variables are declared as local variables. Local variables are declared as given below :

LCL <&variable name> [, <&variable name> ...]

An expansion time variable can be manipulated through the statement SET. The SET statement is written as :

<Expansion time variable> SET <expression>

- In many macros, similar statements are generated during expansion.
- For example, the macro shown in Fig. 3.4.2 generates similar statements.

```
        MACRO
CLEAR   &ARG
MOVER   AREG, = '0'
MOVEM   AREG, &ARG
MOVEM   AREG, &ARG+1
MOVEM   AREG, &ARG+2
MOVEM   AREG, &ARG+3
MEND
```

Fig. 3.4.2 : A macro with similar statements

A call to macro CLEAR with the statement,

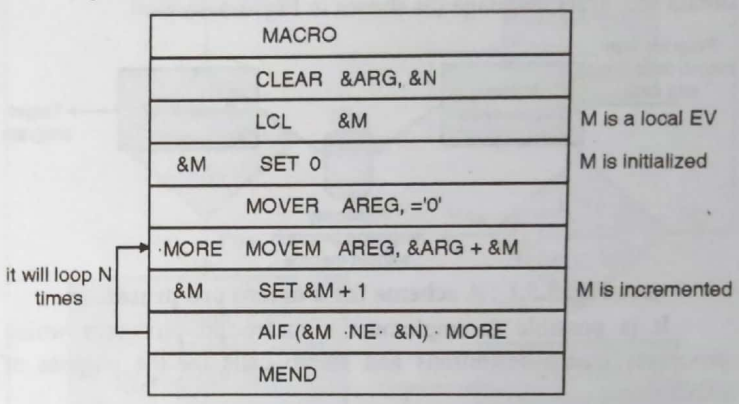
CLEAR A

will lead to following expansion

```
MOVER AREG, = '0'
MOVEM AREG, A
MOVEM AREG, A + 1
MOVEM AREG, A + 2
MOVEM AREG, A + 3
```

The above code stores the value '0' in four consecutive locations with the address A, A + 1, A + 2 and A + 3.

- Alternatively, the same effect can be created by implementing loop for expansion. Loop can create the same effect as given in the macro CLEAR. Expansion time loop can be written using expansion time variable.
- The macro given in Fig. 3.4.2 can be re-written as shown in Fig. 3.4.3.



(S3.5)**Fig. 3.4.3 : An equivalent macro written using expansion time variable (EV)**

- A call to macro CLEAR (Fig. 3.4.3) with the statement.

CLEAR A, 3

will loop three times for each value of M from 0 to 3 with the following expansion.

```
MOVER AREG, = '0'
MOVEM AREG, A      ——  M = 0
MOVEM AREG, A + 1  ——  M = 1
MOVEM AREG, A + 2  ——  M = 2
MOVEM AREG, A + 3  ——  M = 3
```

Example 3.4.1

Can a one pass macro processor successfully handle a macro containing macro pseudo-ops ? If not what modifications are necessary to enable it to handle such situations ? If yes, how does it handle ?

Solution :

A single pass macro processor requires that macro definitions must be given before they are called. Macro definitions must be processed before calls.

Macro processor pseudo-ops, AIF and AGO, permit conditional reordering of the sequence of macro expansion. These details can be stored in macro definition table in an intermediate form. A one pass macro processor can successfully handle a macro containing macro pseudo-ops.

Handling of pseudo-ops is discussed in section 3.4.

Scanned with CamScanner

Syllabus Topic : Design of a Two-pass Macro-processor

3.5 Design of Macro Processor

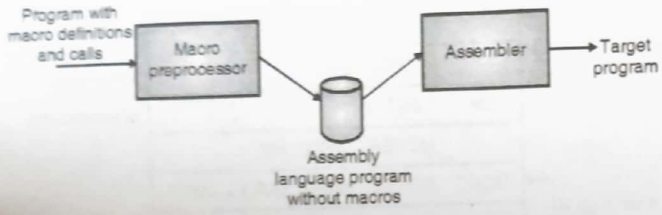
→ (May 2013, May 2014)

Q. List down the steps in designing a Macro Preprocessor. **SPPU- May 2013, 6 Marks**

Q. Explain the process of Macro Expansion with relevant data structures. **SPPU - May 2013, 8 Marks**

Q. Explain design of two pass MACRO processor in detail. **SPPU - May 2014, 8 Marks**

Macro pre-processor takes a source program containing macro definitions and macro calls and translates into an assembly language program without any macro definitions or calls. This program can now be handed over to a conventional assembler to obtain the target language (as shown in Fig. 3.5.1).



(S3.6) **Fig. 3.5.1 : A scheme for a macro pre-processor**

It is possible to implement a macro pre-processor which processes macro definitions and macro calls for the purpose of expansions.

3.5.1 Issues Related to the Design of a Simple Macro Preprocessor

We will go for a simple two pass macro pre-processor and then enhance it to handle advance features like :

1. AIF
2. AGO
3. Sequencing symbol
4. Expansion time variable

The macro pre-processor has to perform the four basic tasks :

1. Recognize macro definition
2. Save the macro definition
3. Recognize macro calls
4. Expand macro calls and substitute arguments.

- A macro definition is identified by MACRO and MEND pseudo opcodes.
- A macro definition is saved as it is required during macro expansion.
- A macro call appears as operation mnemonic.
- During macro expansion, the pre-processor must substitute formal parameters with actual parameters.

A macro pre-processor has to do the following :

Pass 1

Scan all macro definitions one by one. For each macro:

(a) Enter its name in the Macro Name Table (MNT).

(b) Store the entire macro definition in the Macro Definition Table (MDT).

(c) Add the information to the MNT indicating where the definition of a macro can be found in MDT.

(d) Prepare argument list array.

Pass 2

Examine all statements in the assembly source program to detect macro calls. For each macro call :

(a) Locate the macro name in MNT.

(b) Establish correspondence between formal parameters and actual parameters.

(c) Obtain information from MNT regarding position of the macro definition in MDT.

(d) Expand the macro call by picking up model statements from MDT.

Example 3.5.1

Consider the following code segment.

```
MACRO
INCR        &X, &Y, &REG = AREG
MOVER       &REG, &X
ADD         &REG, &Y
MOVEM       &REG, &X
MEND
MACRO
DECR        &A, &B, &REG = BREG
MOVER       &REG, &A
SUB         &REG, &B
MOVEM       &REG, &A
MEND
START       100
READ N1
READ N2
INCR  N1, N2, REG = CREG
DECR N1, N2
STOP
N1          DS 1
N2          DS 1
END
```

Show the contents of

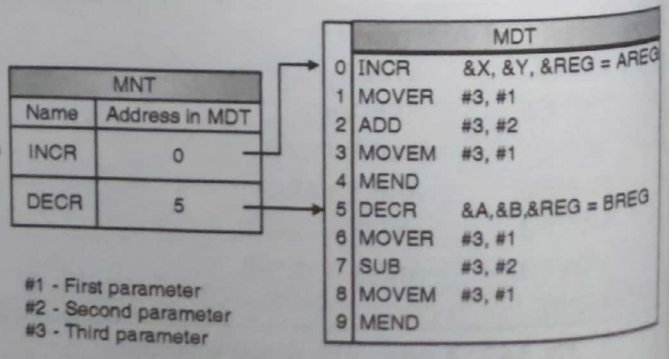
(i) Macro Name Table

(ii) Macro Definition Table

(iii) Argument List Array

Solution :

Pass I of the macro pre-processor will store the details of the two macros in MNT and MDT.



MNT

	Name	Address in MDT
0	INCR	0
1	DECR	5

MDT

0	INCR	&X, &Y, &REG = AREG
1	MOVER	#3, #1
2	ADD	#3, #2
3	MOVEM	#3, #1
4	MEND	
5	DECR	&A,&B,&REG = BREG
6	MOVER	#3, #1
7	SUB	#3, #2
8	MOVEM	#3, #1
9	MEND	

#1 - First parameter
#2 - Second parameter
#3 - Third parameter

(S3.7) **Fig. Ex. 3.5.1**

Scanned with CamScanner

Pass II of the macro pre-processor will create the argument list array, every time there is a call to macro, and expand the macro.

(1) Macro call :

```
INCR    N1, N2, REG = CREG
```

Argument list array

N1
N2
CREG

Expanded code :

```
MOVER   CREG, N1
ADD     CREG, N2
MOVEM   CREG, N1
```

(2) Macro call :

```
DECR    N1, N2
```

Argument list array

N1	
N2	
AREG	–Default value

Expanded code :

```
MOVER   AREG, N1
SUB     AREG, N2
MOVEM   AREG, N1
```

3.5.2 Databases used in Pass-1 of 2 Pass Macro Processor

➔ **(Dec. 2013)**

Q. What are the data structure used for the design of macro processing ? **SPPU - Dec. 2013, 3 Marks**

Pass 1

Uses the following databases :

1. Source program as input
2. Source program without macro definition as output of pass1 and input of pass2.
3. Macro definition table (MDT)
4. Macro name table (MNT)
5. Argument list array (ALA)
6. MNTP (macro name table pointer)
7. MDTP (macro definition table pointer)

A source program containing both macro definitions and macro calls is given as input to pass1 of microprocessor.

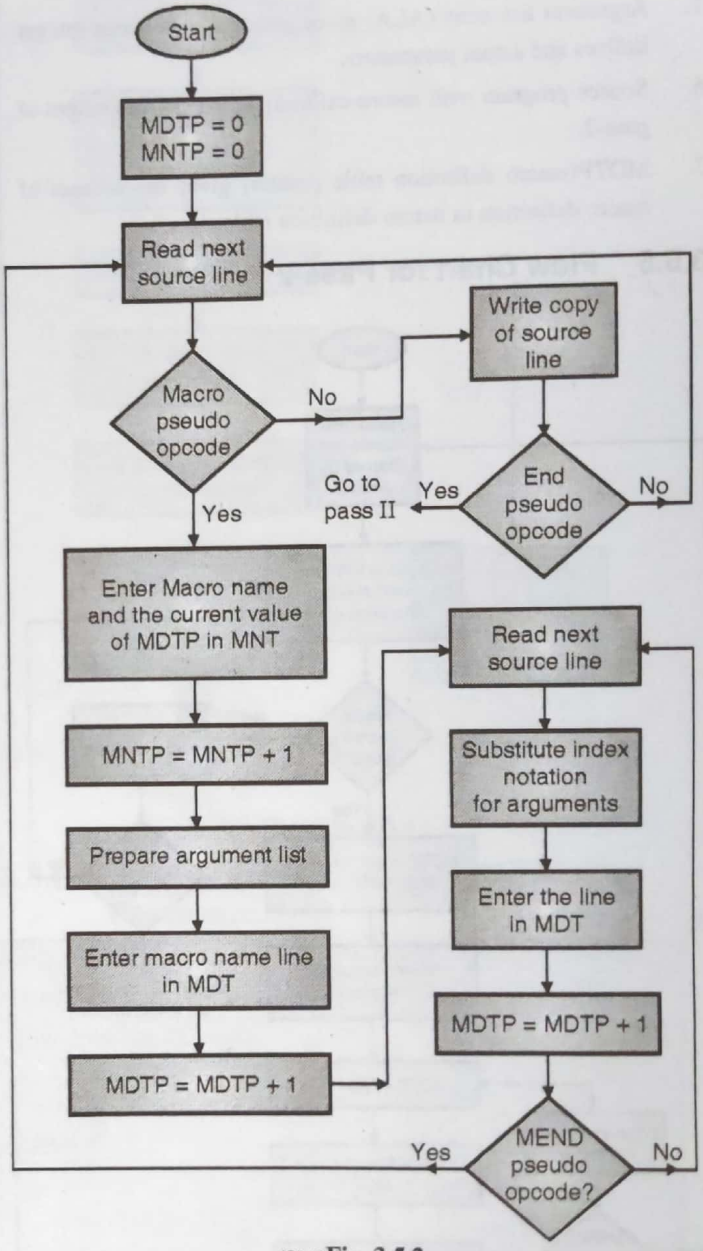
The MNT is used to store name of macros. The entire macro definition is stored in the MDT. The index into MDT (starting row number of this macro definition) is stored in MNT.

The argument list array (ALA) is used to substitute index markers for formal parameters before storing macro definition in MDT.

MNTP gives the next available entry in MNT.

MDTP gives the next available entry in MDT.

3.5.3 Flow Chart for Pass 1



(S3.8)**Fig. 3.5.2**

3.5.4 Databases used in Pass-2 of 2 Pass Macro Processor

➔ **(Dec. 2013)**

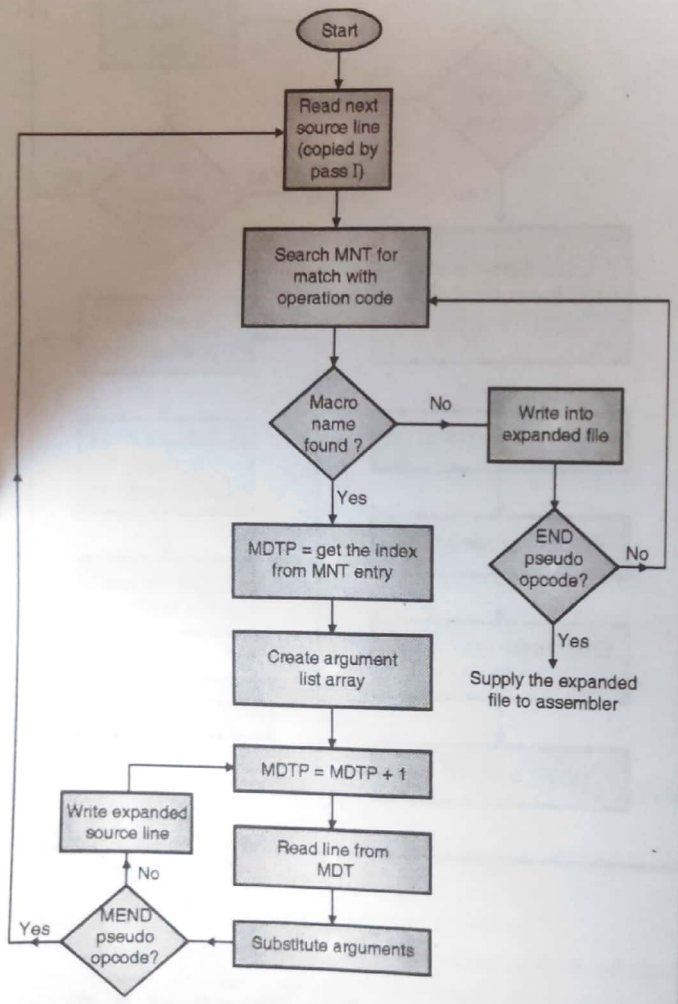
Q. What are the data structure used for the design of macro processing? **SPPU - Dec. 2013, 3 Marks**

Scanned with CamScanner

Pass-2 uses the following databases

1. Input source program for pass-2. It is produced by pass-1.
2. Macro definition table (MDT) produced by pass-1.
3. Macro name table (MNT) produced by pass-1.
4. MNTP (macro name table pointer) gives number of entries in MNT.
5. Argument list array (ALA) gives association between integer indices and actual parameters.
6. Source program with macro-calls expanded. This is output of pass-2.
7. MDTP(macro definition table pointer) gives the address of macro definition in macro definition table.

3.5.5 Flow Chart for Pass-2



(S3.9)**Fig. 3.5.3**

3.5.6 A Simple One-Pass Macro Processor

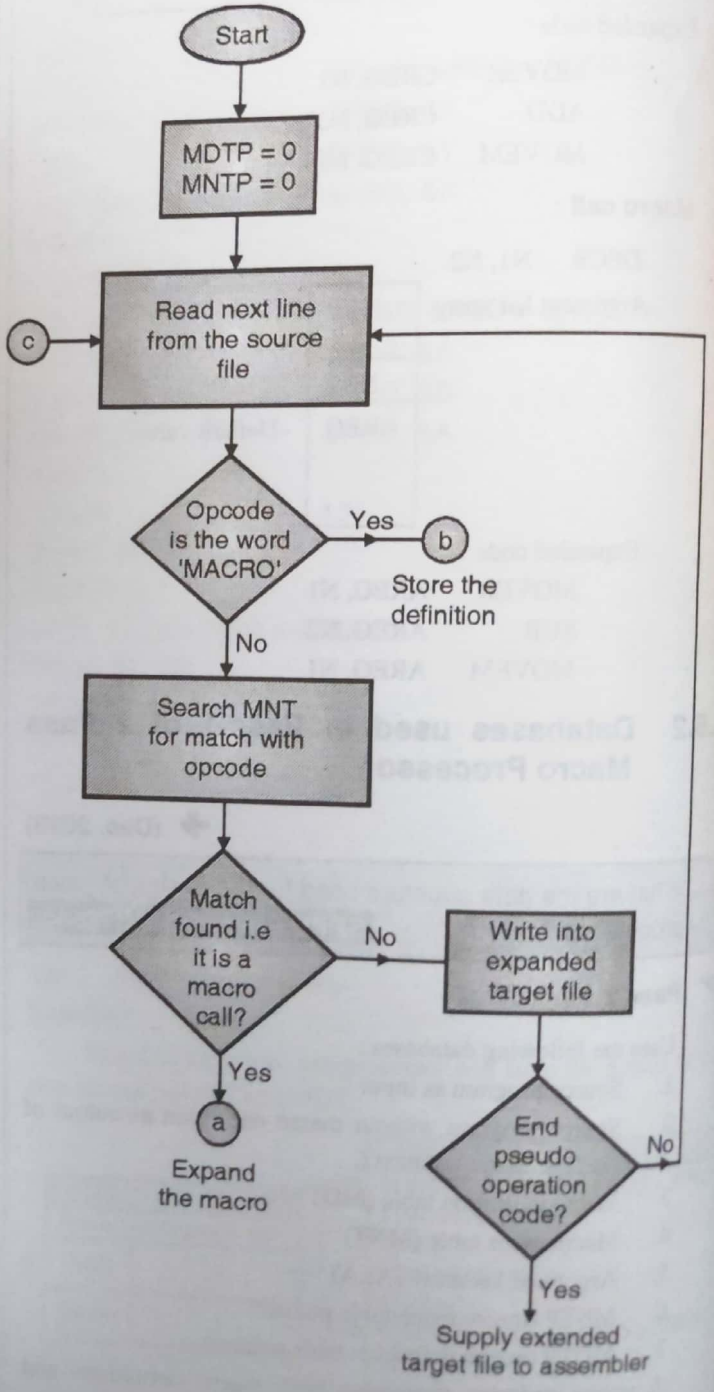
We can write a single-pass macro processor if the macro definition appears before macro calls. Single-pass macro processors are more efficient. We can combine two passes into a single pass by :

(1) Storing definition of a macro in MDT, when a macro definition appears in the source file.

(2) Expanding a macro, when a macro call appears in the source file.

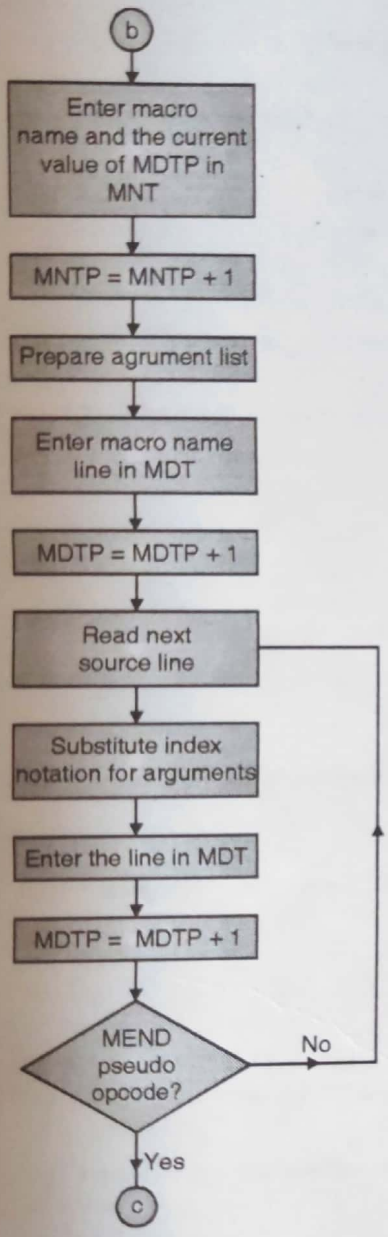
(3) Other lines from the source file are copied as it is into the target (expanded) file.

A flow chart for one-pass macro processor is shown in Fig. 3.5.4.

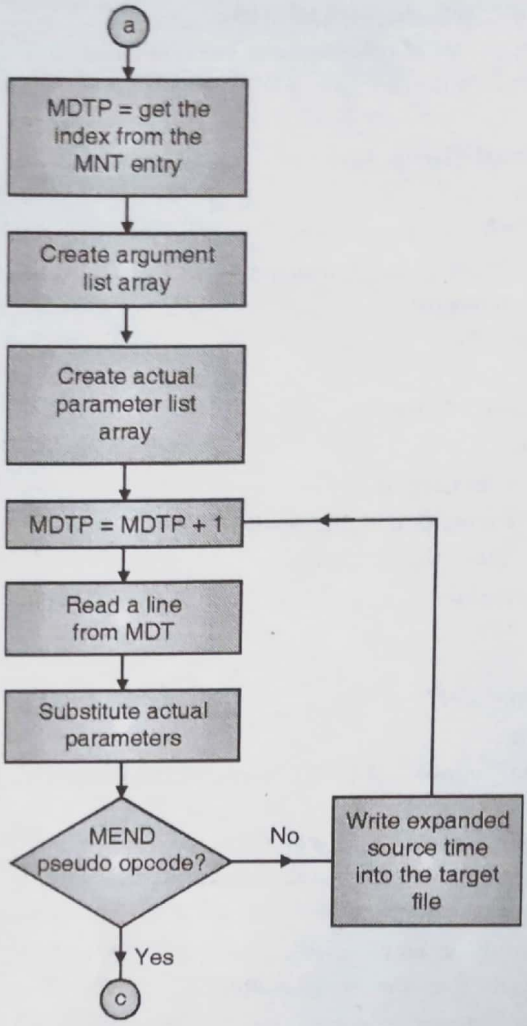


(S3.10)**Fig. 3.5.4 Contd.....**

Scanned with CamScanner



(S3.10)**Fig. 3.5.4 Cont...**



(S3.10)**Fig. 3.5.4**

3.5.7 C-Program for One-pass Macro Processor

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>

struct MNT
{
    char mname[20];
    int  mdtp;
}mnt[5];

struct MDT
{
    char opcode[15],rest[35];
}mdt[30];

char arglist[15][15],apt[10][15];
int mdtp=0,mntp=0,arglistp=0;
char FName[20], TName[20];
```

Scanned with CamScanner

```
char Buffer[80], temp[40],tok1[40];
int pp,kpp;// no. of positional and keyword parameters
FILE *fp1, *fp2;

int SearchMNT(char *s)
{   int i;
    for(i=0; i<mntp; i++)
       if(strcmpi(s,mnt[i].mname)==0)
          return(i);
    return(-1);
}
int SearchPNT(char *s)
{   int i;
    for(i=0;i<arglistp;i++)
       if(strcmpi(arglist[i],s)==0)
          return(i);
    return(-1);
}

void Print_MNT()
{   int i;
    printf("\n\n------MACRO NAME TABLE----------
");
    printf("\n#\tMName\t#MDTP");
    printf("\n-----------------------------------------");
    for(i=0;i<mntp;i++)
       printf("\n%d\t%s\t%d",
       i,mnt[i].mname,mnt[i].mdtp);
    printf("\n-----------------------------------------");
    getch();
}

void Print_PNT()
{   int i;
    printf("\n\n------PARAMETER NAME TABLE--------");
    printf("\n#\tPName");
    printf("\n-------------------------------");
    for(i=0;i<arglistp;i++)
       printf("\n%d\t%s\t%s",i,arglist[i],apt[i]);
    printf("\n-------------------------------");
    getch();
}

void Print_MDT()
{   int i;
    printf("\n\n-------MACRO DEFINITION TABLE--------
");
    printf("\nOpcode\tRest");
    printf("\n-----------------------------------------");
    for(i=0;i<mdtp;i++)
       printf("\n%d\t%s\t%s",i,mdt[i].opcode,mdt[i].rest);
    printf("\n-----------------------------------------");
    getch();
}
char *nexttoken(char *str , char *token)
  {
     int i;
     while(*str==' ')
     str++;
     if(*str==',' || *str=='=')
     {
       *token=*str;
       token++;
       str++;
       *token='\0';
       return(str);
     }
     while(isalnum(*str) || *str=='#' || *str=='&')
     {
       *token=*str;
       token++;
       str++;
     }
    *token='\0';
    return(str);
  }

void make_arglist(char *s)
{
    int k;
    pp=kpp=0; //no. of positional and keyword
parameters
    arglistp=0;
    while(*s)
    {
      s=nexttoken(s,temp);
      k = SearchPNT(temp+1);
      if(k==-1)
         strcpy(arglist[arglistp++],temp+1);
      else
      {
      printf("\nError: Multiple Declaration of Symbol %s in
         Argument List",temp);
            exit(0);
      }
      s=nexttoken(s,temp);
      if(*temp=='=')
        {   kpp++;
         break; //handle keyword parameter
        }
      pp++;
    }

    if(*temp=='=')          //handle keyword parameter
```

Scanned with CamScanner

```
        while(*s)
        {
          s=nexttoken(s,temp);
          if(*temp != ',' )
              strcpy(apt[arglistp-1],temp);

          else
              strcpy(apt[arglistp-1],"");
          s=nexttoken(s,temp);
          if(*temp == '\0')
            break;
          k = SearchPNT(temp+1);
          if(k == -1)
              strcpy(arglist[arglistp++],temp+1);
          else
          {
              printf("\nError:  Multiple  Declaration  of
Symbol
                  %s in Argument List",temp);
              exit(0);
          }
        }

  }

void Expand(int n)
{
    int a;
    int MEC;
    char *pointer;
    MEC = mnt[n].mdtp+1;
    while(strcmpi(mdt[MEC].opcode,"MEND")!= 0)
    {          fprintf(fp2,"+%s  ",mdt[MEC].opcode);
        pointer = mdt[MEC].rest;
        pointer = nexttoken(pointer,tok1);
        while(tok1[0]!='\0')
        {
          if(tok1[0] == '#')
            {
              a = atoi(tok1+1);
              fprintf(fp2,"%s", apt[a]);
            }
          else
              fprintf(fp2,"%s",tok1);
          pointer = nexttoken(pointer,tok1);
        }
        fprintf(fp2,"\n");
        MEC++;
    }
}

void main()
{
    int i=0,j=0,k=0,n;
    char *pointer ;//pointer for the array buffer
    clrscr();
    printf("\nEnter Source File Name: ");
    scanf("%s",FName);
    printf("\nEnter Target File Name: ");
    scanf("%s",TName);
    if((fp1=fopen(FName,"r"))==NULL)
    {     printf("\nCannot Open Source File...%s",FName);
      exit(0);
    }
    if((fp2=fopen(TName,"w"))==NULL)
    {
     printf("\nCannot Create Intermediate File...%s",TName);
     exit(0);
    }
    while(fgets(Buffer,80,fp1))
    {
      pointer = Buffer;
      nexttoken(pointer,tok1);
      if(strcmpi(tok1,"MACRO") == 0)
      {
        fgets(Buffer,80,fp1); //read the parameter line
        pointer=nexttoken(pointer,tok1);
        strcpy(mnt[mntp].mname,tok1);
        mnt[mntp].mdtp = mdtp;
        strcpy(mdt[mdtp].opcode,tok1);
        strcpy(mdt[mdtp].rest,pointer);
        mdtp++;
        make_arglist(pointer);
        mntp++;
       while(fgets(Buffer,80,fp1))
    //store the body of the macro
        {
        pointer=Buffer;
        pointer=nexttoken(pointer,tok1);
        if(strcmpi(tok1,"MEND")==0)
        {
            strcpy(mdt[mdtp].opcode,"MEND");
            strcpy(mdt[mdtp++].rest,"");
            arglistp=0;;
            break;
        }
        else
        {
          strcpy(mdt[mdtp].opcode,tok1);
          strcpy(mdt[mdtp].rest,"");
          pointer=nexttoken(pointer,tok1);
          while(tok1[0]!='\0')
          {
```

Scanned with CamScanner

```
        if(tok1[0]=='&')
          {
          k = SearchPNT(tok1+1);
            if(k==-1)
            {
              printf("\nError: Parameter %s not
              found",tok1+1);
              exit(0);
            }
            temp[0]='#';
            temp[1]=k+48;//convert to ASCII
            temp[2]='\0';
            strcat(mdt[mdtp].rest,temp);
      }
        else
            strcat(mdt[mdtp].rest,tok1);
        pointer=nexttoken(pointer,tok1);

      }
    }
    mdtp++;
  }

}
else
  {
  k = SearchMNT(tok1);
  if(k==-1)
  fprintf(fp2,"%s",Buffer);
  else
   {
      arglistp=0;
      pointer=mdt[mnt[k].mdtp].rest;
      make_arglist(pointer);
      pointer=nexttoken(pointer,tok1);
      //Handle positional parameters
      pointer=Buffer;
      pointer=nexttoken(pointer,tok1);//skip macro

      name
      for(i=0;i<pp;i++)
       {
      pointer=nexttoken(pointer,tok1);
      strcpy(apt[i],tok1);
      pointer=nexttoken(pointer,tok1);//skip ,
     }
    //Handle keyword parameters
    pointer=nexttoken(pointer,tok1);
    while(tok1[0]!='\0')
     {
        j=SearchPNT(tok1);
      //get location of the keyword parameter
        //get the new value of the keyword parameter
        pointer=nexttoken(pointer,tok1);//skip =
        pointer=nexttoken(pointer,tok1);
        strcpy(apt[j],tok1);
        pointer=nexttoken(pointer,tok1);//skip ,
        pointer=nexttoken(pointer,tok1);
      //read next
          parameter
       }

      Print_PNT();
      Expand(k);
    }//macro expansion
  }
}
Print_MNT();
Print_MDT();
fcloseall();
}
```

Input / Output

```
Enter Source File Name: test.asm

Enter Target File Name: test.ini


------PARAMETER NAME TABLE--------
#     PName
-------------------------------
0     X     N1
1     Y     N2
2     REG   CREG
-------------------------------

------PARAMETER NAME TABLE--------
#     PName
-------------------------------
0     A     N1
1     B     N2
2     REG   BREG
-------------------------------

---------MACRO NAME TABLE----------
#     MName  #MDTP
-----------------------------------
0     INCR   0
```

Scanned with CamScanner

```
1    DECR  5
-----------------------------------------

----------MACRO DEFINITION TABLE----------
Opcode Rest
-----------------------------------------
0    INCR   &X, &Y,&REG=AREG

1    MOVER  #2,#0
2    ADD    #2,#1
3    MOVEM  #2,#0
4    MEND
5    DECR   &A,&B,&REG=BREG

6    MOVER  #2,#0
7    SUB    #2,#1
8    MOVEM  #2,#0
9    MEND
-----------------------------------------
```

Contents of Input File

```
MACRO
INCR &X, &Y, &REG = AREG
MOVER &REG, &X
ADD &REG, &Y
MOVEM &REG, &X
MEND
MACRO
DECR &A,&B,&REG = BREG
MOVER &REG, &A
SUB &REG, &B
MOVEM &REG, &A
MEND
START 100
READ N1
READ N2
INCR N1, N2, REG = CREG
DECR N1, N2
STOP
N1 DS 1
N2 DS 1
END
```

Contents of expanded file

```
START 100
READ N1
READ N2
+MOVER  CREG,N1
+ADD  CREG,N2
+MOVEM  CREG,N1
+MOVER  BREG,N1
+SUB  BREG,N2
+MOVEM  BREG,N1
STOP
N1 DS 1
N2 DS 1
END
```

3.6 Handling of Nested Macro Calls

➔ **(Dec. 2013, May 2014)**

Q. How to handle macro cells within macros ?
SPPU - Dec. 2013, 6 Marks

Q. Explain the term nested macro calls.
SPPU - May 2014, 2 Marks

There are several methods for handling of nested macro calls. These methods are :

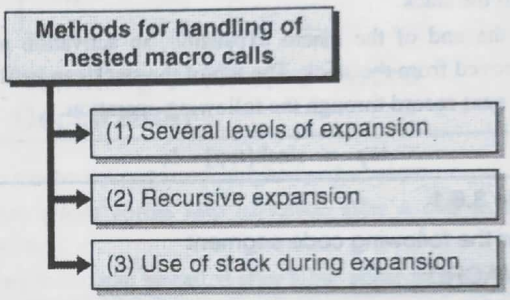


Fig. C3.2 : Methods for handling of nested macro calls

➔ **(1) Several levels of expansions**

Fig. 3.6.1 illustrates nested macro calls. The macro call

COMPUTE1 X, Y, Z

Call be expanded (level 1) using the algorithm of macro expansion.

$$\text{COMPUTE1 X, Y, Z} \Rightarrow \begin{Bmatrix} \text{COMPUTE} & \text{X} \\ \text{COMPUTE} & \text{Y} \\ \text{COMPUTE} & \text{Z} \end{Bmatrix}$$

The expanded code itself contains macro calls. The macro expansion algorithm can be applied to the first level expanded code to expand these macro calls and so on, until we obtain a code form which does not contain any macro calls. This approach requires several passes of expansion, which is not desirable.

➔ **(2) Recursive expansion**

To handle nested macro calls, the macro expansion function should be able to work recursively. During recursion, while processing one macro the processing of inner macro can begin and after the expansion of inner macro finishes, the processing of outer macro may continue. A recursion is handled through a stack, where local variables are stored onto the stack before making a recursive call.

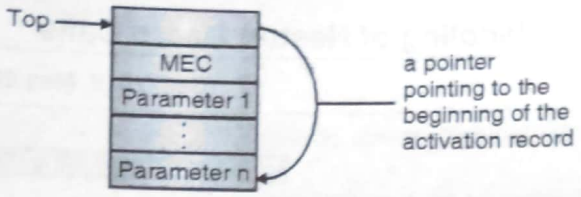
➔ **(3) Use of stack during expansion**

Nested macro calls can be handled with the help of explicit stack.

- Macro calls are handled in LIFO manner.
- Stack can be used to accommodate the expansion time data structure.

Scanned with CamScanner

- Expansion time data structure include :
  1. MEC – Macro expansion counter
  2. Actual parameter table
- Expansion time data structure is stored in an activation record. The structure of the activation record is given in Fig. 3.6.1.



(S3.11)**Fig. 3.6.1**

- Every call to a macro involves pushing an activation record onto the stack.
- At the end of the macro expansion, an activation record is removed from the stack. The top of the stack can be shifted to the next record through the following operation.

```
top = stack[top] – 1;
```

Example 3.6.1

Consider the following code segment

```
1.  MACRO
2.  INCR    &A, &B, &REG
3.  MOVER   &REG, &A
4.  ADDS    &A, &B
5.  MOVEM   &REG, &A
6.  MEND
7.  MACRO
8.  ADDS    &F, &S
9.  MOVER   AREG, &F
10. ADD     AREG, &S
11. MOVEM   AREG, &S
12. WRITE   &S
13. MEND
14. MACRO
15. SUBS    &F, &S
16. MOVER   BREG, &F
17. SUB     BREG, &S
18. MOVEM   BREG, &S
19. WRITE   &S
20. MEND
21. START   200
22. READ    N1
23. READ    N2
24. ADDS    N1, N2
25. SUBS    N1, N2
26. INCR    N1, N2, DREG
27. STOP
28. N1      DS  2
29. N2      DS  2
```

Show the content of

(i) Macro name table

(ii) Macro definition table

(iii) Argument list Array

Solution :

Pass I : Contents of MNT and MDT at the end of Pass I.

	MNT	
	Name	**MDTP**
0	INCR	0
1	ADDS	5
2	SUBS	11

	MDT	
	Opcode	**Rest**
0	INCR	&A, &B, &REG
1	MOVER	#2, #0
2	ADDS	#0, #1
3	MOVEM	#2, #0
4	MEND	
5	ADDS	&F, &S
6	MOVER	AREG, #0
7	ADD	AREG, #1
8	MOVEM	AREG, #1
9	WRITE	#1
10	MEND	
11	SUBS	&F, &S
12	MOVER	BREG, #0
13	SUB	BREG, #1
14	MOVEM	BREG, #1
15	WRITE	#1
16	MEND	

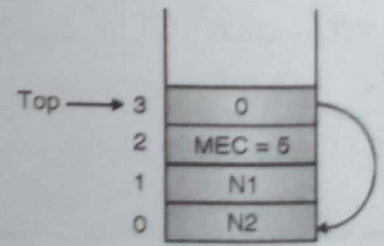
Pass I

1. Expansion of line 24 ADDS N1, N2

Argument list array :

0	N1
1	N2

Activation record on the stack :



Expanded Code :

```
MOVER   AREG,  N1
ADD     AREG,  N2
MOVEM   AREG,  N2
WRITE   N2
```

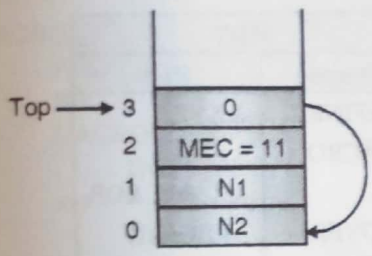
2. Expansion of line 25 SUBS N1, N2

Argument list array :

0	N1
1	N2

Scanned with CamScanner

Activation record on the stack :



Expanded Code :

```
MOVER   BREG,  N1
SUB     BREG,  N2
MOVEM   BREG,  N2
WRITE   N2
```

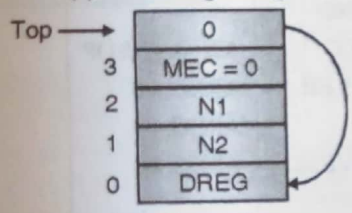
3. Expansion of line 26 INCR N1, N2, DREG

Argument list array :

0	N1
1	N2
2	DREG

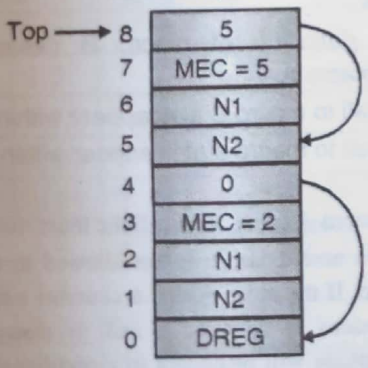
Activation record (stack)

(a) At the beginning



(b) At the time of nested macro call

ADDS #0, #1



Expanded Code :

```
MOVER   DREG,  N1
MOVER   AREG,  N1
ADD     AREG,  N2
MOVEM   AREG,  N2
WRITE   N2
MOVEM   DREG,  N1
```

Expanded source file at the end of pass II :

```
START   200
READ    N1
READ    N2
MOVER   AREG,  N1
ADD     AREG,  N2
MOVEM   AREG,  N2
WRITE   N2
```

Expansion of ADDS N1, N2

```
MOVER   BREG,  N1
SUB     BREG,  N2
MOVEM   BREG,  N2
WRITE   N2
```

Expansion of SUBS N1, N2

```
MOVER   DREG,  N1
MOVER   AREG,  N1
ADD     AREG,  N2
MOVEM   AREG,  N2
WRITE   N2
MOVEM   DREG,  N1
STOP
N1      DS  2
N2      DS  2
```

Expansion of INCR N1, N2, DREG

3.7 Handling of Nested Macro Declaration

A macro can be defined inside the body of a macro.

- Inner macro comes into existence after a call to the outer macro.
- Inner macro can be called after it has come into existence.

Example 3.7.1

Consider the following program segment :

```
MACRO
DEFINE      &XYZ
MACRO
&XYZ        &X, &Y, &OP
MOVER       AREG, &X
&OP         AREG, &Y
MOVEM       AREG, &X
MEND
MEND
MACRO
COMPUTE     &F, &S
MOVEM       BREG, TMP
INCRM       &F, &S, BREG
MOVER       BREG, TMP
MEND
MACRO
INCRM       &M, &I, &R
MOVER       &R, &M
ADD         &R, &I
MOVEM       &R, &M
MEND
START       100
DEFINE      CACL
COMPUTE     X, Y
CALC        A, B, MULT
END
```

(i) Show the contents of MDT and MNT after macro processing

(ii) Expanded assembly language program.

Scanned with CamScanner

Solution :

(i) MDT and MNT

MNT	
Name	**MDTP**
DEFINE	0
COMPUTE	8
INCRM	13
CALC	18

	MDT	
	Opcode	**Rest**
0	DEFINE	&XYZ
1	MACRO	
2	#0	&X, &Y, &OP
3	MOVER	AREG, &X
4	&OP	AREG, &Y
5	MOVEM	AREG, &X
6	MEND	
7	MEND	
8	COMPUTE	&F, &S
9	MOVEM	BREG, TMP
10	INCRM	#0, #1, BREG
11	MOVER	BREG, TMP
12	MEND	
13	INCRM	&M, &I, &R
14	MOVER	#2, #0
15	ADD	#2, #1
16	MOVEM	#2, #0
17	MEND	
18	CALC	&X, &Y, &OP
19	MOVER	AREG, #0
20	#2	AREG, #1
21	MOVEM	AREG, #0
22	MEND	

This will come into existence after a call to DEFINE

(ii) Expanded assembly language program

Source line		Expanded code
START 100	–	START 100
DEFINE CALC	–	NO code will be generated
COMPUTE X, Y	–	MOVEM BREG, TMP
		INCRM &M, &I, &R

	–	MOVER BREG, X
		ADD BREG, Y
		MOVEM BREG, X

		MOVER BREG, TMP
CALC A, B, MULT	–	MOVER AREG, A
		MULT AREG, B
		MOVEM AREG, A
END	–	END

Thus the final code will be

```
        START   100
MOVEM   BREG,   TMP
MOVER   BREG,   X
ADD     BREG,   Y
MOVEM   BREG,   X
MOVER   BREG,   TMP
MOVER   AREG,   A
MULT    AREG,   B
MOVEM   AREG,   A
END
```

Example 3.7.2

Consider the definition of macro B nested within the definition of a macro A.

(i) Can a call to macro B also appear within macro A ?

(ii) Can a call to macro a also appear within macro B ?

Solution :

(i) An outer macro A can always call the inner macro B.

(ii) Normally, a nested macro is not allowed to make a call to outer macro. If the outer macro A contains a call to macro B and the macro B contains a call to macro A then the expansion phase will be caught in a peculiar situation where it will never end.

Example 3.7.3

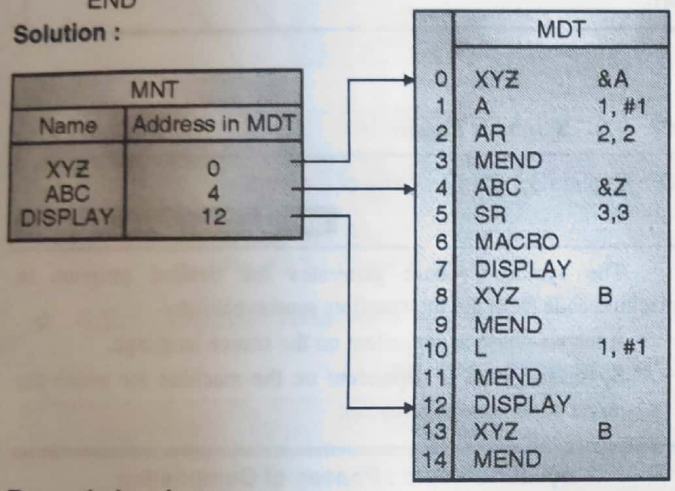
Consider the following code, show the contents of macro name table and macro definition table.

```
START   100
SR      2, 2
USING   *, 15
MACRO
    XYZ     & A
    A       1, & A
    AR      2, 2
```

Scanned with CamScanner

```
        MEND
        L       1, D1
        MACRO
            ABC     &z
            SR      3, 3
            MACRO
                DISPLAY
            xyz     B
                MEND
            L       1, & z
            MEND
            xyz     B1
            SR      4, 4
            ABC     B1
D1  DC          F'4'
B1  DC          F'5'
    END
```

Solution :



Expanded code

	START	100	
	SR	2, 2	
	USING	*, 15	
	L	1, D1	
+	A	1, B1	xyz B1 is expanded
+	AR	2, 2	
	SR	4, 4	
+	SR	3, 3	ABC B1 is expanded
+	L	1, B1	
D1	DC	F '4'	
B1	DC	F '5'	
	END		

3.8 Exam Pack (University Questions)

☞ **Syllabus Topic : Macro Definition**

Q. Explain the term macro definition.
(Refer section 3.1) (2 Marks) **(May 2014)**

☞ **Syllabus Topic : Macro Call**

Q. Explain the term macro calls.
(Refer section 3.2.1) (2 Marks) **(May 2014)**

☞ **Syllabus Topic : Macro Expansion**

Q. Explain the process of Macro Expansion with relevant data structures.
(Refer section 3.2.2) (8 Marks) **(May 2013)**

Q. Explain the term macro expansion.
(Refer section 3.2.2) (2 Marks) **(May 2014)**

Example 3.2.1 (6 Marks) **(Oct. 2016 (In Sem))**

Q. Explain the advance macro facilities : Attributes of parameters. *(Refer section 3.2.3)* **(Dec. 2015)**

☞ **Syllabus Topic : Nested Macro Calls**

Q. Explain nested macros with example.
(Refer section 3.3) (3 Marks)
(Dec. 2014, May 2015, May 2016)

☞ **Syllabus Topic : Advanced Macro Facilities**

Q. Explain the process of alteration of flow of control during macro expansion.
(Refer section 3.4) (4 Marks) **(May 2013)**

Q. Explain the advance macro facilities : Alteration of flow of control during expansion.
(Refer section 3.4) (3 Marks) **(Dec. 2015)**

Q. Explain advanced macro facilities with example.
(Refer section 3.4) (6 Marks) **(May 2017)**

Q. Explain expansion time variables with example.
(Refer section 3.4.1) (3 Marks)
(Dec. 2014, May 2015)

Q. Explain the advance macro facilities : Expansion time variables *(Refer section 3.4.1)* (3 Marks)
(Dec. 2015)

☞ **Syllabus Topic : Design of a Two-pass Macro-processor**

Q. List down the steps in designing a Macro Preprocessor. *(Refer section 3.5)* (6 Marks)
(May 2013)

Q. Explain the process of Macro Expansion with relevant data structures.
(Refer section 3.5) (8 Marks) **(May 2013)**

Q. Explain design of two pass MACRO processor in detail. *(Refer section 3.5)* (8 Marks) **(May 2014)**

Q. What are the data structure used for the design of macro processing ?
(Refer sections 3.5.2 and 3.5.4) (6 Marks)
(Dec. 2013)

Q. How to handle macro cells within macros ?
(Refer section 3.6) (6 Marks) **(Dec. 2013)**

Q. Explain the term nested macro calls.
(Refer section 3.6) (2 Marks) **(May 2014)**

❑❑❑

Scanned with CamScanner

UNIT II

CHAPTER 4

Compilers

Syllabus Topics

Basic compilers function, Phases of compilation, memory allocation, Compilation of expressions, compilation of control structures, Code of optimization.

Syllabus Topic : Basic Compliers Function

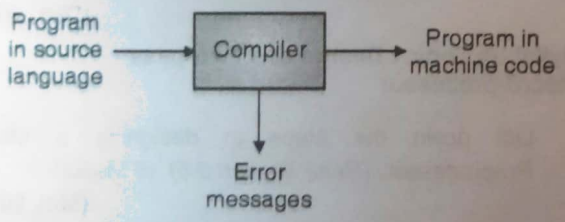
4.1 Basic Compilers Function

➔ (Dec. 2013, Dec. 2014)

Q. Explain : Complier.

SPPU - Dec. 2013, Dec. 2014, 4 Marks

A compiler is a program that reads a program written in source language and translates it into an equivalent program in machine code. During compilation, it reports the presence of errors in the source program.



(S4.1) **Fig. 4.1.1 : A Compiler**

Compilation activity can be broken down into two parts :

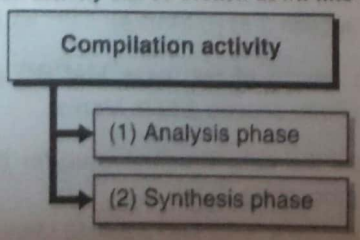


Fig. C4.1 : Compilation activity

➔ **(1) Analysis Phase**

Analysis phase reads the source program and creates an intermediate representation of the source program. Intermediate form of representation is independent of both :

(1) Source language (application domain)

(2) Machine language (execution domain)

➔ **(2) Synthesis Phases** ➔ (Dec. 2014)

Q. Explain synthesis phase of a compiler.

SPPU - Dec. 2014, 6 Marks

The synthesis phase generates the desired program in machine code from the intermediate representation.

- Analysis phase is dependent on the source language.
- Synthesis phase is dependent on the machine for which the target code is to be generated.

Syllabus Topic : Phases of Compilation

4.2 Phase of Compilers

➔ (Aug. 2015, May 2017)

Q. Explain phases of a compiler with example.

SPPU - Aug. 2015(In Sem), May 2017, 6 Marks

A compiler works in phases. Output of one phase becomes the input of next phase. Phases of compilers are :

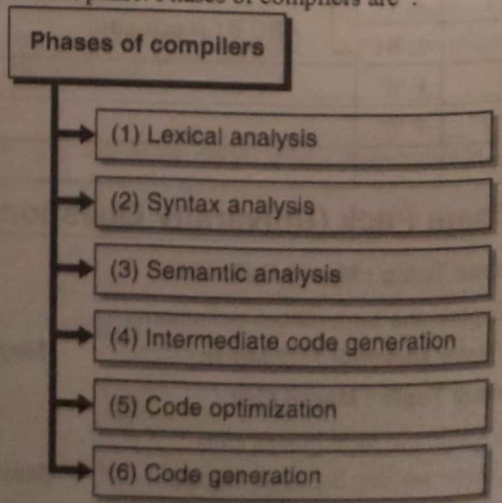
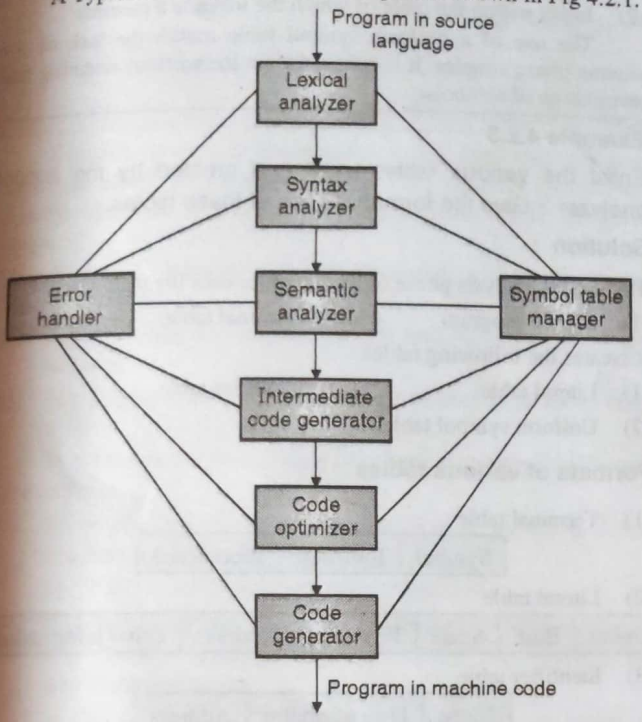


Fig. C4.2 : Phases of compilers

Scanned with CamScanner

A Typical decomposition of a compiler is shown in Fig 4.2.1.



(S4.2)**Fig. 4.2.1 : Phase of a compiler**

4.2.1 Lexical Analysis

➔ **(Dec. 2016, May 2017)**

Q. Explain lexical analysis with example.
SPPU - Dec. 2016, 3 Marks

Q. Explain lexical analysis for language processor.
SPPU - May 2017, 3 Marks

Lexical analysis involves scanning of the source program from left to right and separating them into tokens. A token is a sequence of characters having a collective meaning. Tokens are separated by blanks, operators and special symbols.

A lexical analysis on the statement

$$x = y + z \times 30;$$

will generate the following tokens

x	=	y	+	z	*	30	;

(1) x is an identifier
(2) = is a terminal symbol
(3) y is an identifier
(4) + is a terminal symbol
(5) z is an identifier
(6) * is a terminal symbol
(7) 30 is a literal
(8) ; is a terminal symbol

- Blanks separating the token are eliminated.
- Lexical phase discards comments since they have no effect on the processing of the program.
- Identifiers are stored in the symbol table.
- Literals are stored in the literal table.

Uniform symbol table

This table is created by lexical analysis phase to represent the source program as a stream of tokens rather than of individual characters. Each entry in the uniform symbol table consists of :

(1) Identification of table of which the token is a member. It could be
LIT for literal table.
IDN for identifier table.
TRM for terminal symbol.
(2) Index of the token in the corresponding table.

Table - id	Index

(S4.3)**Fig. 4.2.2 : Uniform symbol table entry**

- Terminal symbol table is a fixed table for any compiler.
- Literal table is generated during lexical analysis.
- Symbol table is generated during lexical analysis.
- Uniform symbol table is generated during lexical analysis.

Example 4.2.1

Consider the following program

```
main ()
{
    int a, b, c, d ;
    char c ;
    a = b + 10 ;
    d = a + b ;
}
```

Write down output of the lexical analyzer for above program. Explain which tables are used (built-in and generated). Show contents of each table.

Solution : Following tables are used during lexical analysis phase.

(1) Terminal table (It is a fixed table).
(2) Symbol table (It is generated during lexical analysis).
(3) Literal table (It is generated during lexical analysis).
(4) Uniform symbol table (It is generated during lexical analysis).

The given program will generate the following tokens :

main	(	)

{

int	a	,	b	,	c	,	d	;

char	c	;

a	=	b	+	10	;

d	=	a	+	b	;

}

Scanned with CamScanner

These tokens are stored in the uniform symbol table. Contents of various tables are shown in Fig. Ex. 4.2.1.

Terminal Table

1	(
2	)
3	{
4	}
5	;
6	=
7	+
8	,
9	int
10	char
	• • •

Symbol Table

	Name	Attributes
1	main	Filled in by later phases
2	a	
3	b	
4	c	
5	d	

Literal Table

Literal	Type	Address
10	int	Filled by later phases

Uniform symbol table

	Class	Index
main	IDN	1
(	TRM	1
)	TRM	2
{	TRM	3
int	TRM	9
a	IDN	2
,	TRM	8
b	IDN	3
,	TRM	8
c	IDN	4
,	TRM	8
d	IDN	5
;	TRM	5
char	TRM	10
c	IDN	4
;	TRM	5
a	IDN	2
=	TRM	6
b	IDN	3
+	TRM	7
10	LIT	1
;	TRM	5
d	IDN	5
=	TRM	6
a	IDN	2
+	TRM	7
b	IDN	3
;	TRM	5

(S4.4) **Fig. Ex. 4.2.1 : Various tables during lexical analysis**

Example 4.2.2

Lexical analysis phase builds a uniform symbol table. What is the need of such a table ?

Solution : Lexical analysis phase of the compiler performs the following tasks :

(1) It parses the source program into the basic elements or tokens of the language.

(2) It builds a literal table and identifier table.

(3) It builds a uniform symbol table.

The uniform symbol table represents the program as a string of tokens rather than of individual characters. Each uniform symbol table entry contains two information :

(1) Identification of the table of which the token is a member.

(2) Index within the table of which the token is a member.

The use of a uniform symbol table makes the task of the parsing phase simpler. It is relieved from the work of scanning and recognition of symbols.

Example 4.2.3

Enlist the various tables used and created by the lexical analyzer ? Give the format of each of these tables.

Solution :

The lexical analysis phase of the compiler uses the following :

(1) Source program (2) Terminal table.

It creates the following tables :

(1) Literal table. (2) Identifier table.

(2) Uniform symbol table.

Formats of various tables

(1) Terminal table

Symbol	Indicator	Precedence

(2) Literal table

Literal	Base	Scale	Precision	Address	Other information

(3) Identifier table

Name	Data attributes	Address

(4) Uniform symbol table

Table	Index

4.2.2 Syntax Analysis

(Dec. 2016, May 2017)

Q. Explain syntax analysis with example.

SPPU - Dec. 2016, 3 Marks

Q. Explain syntax analysis for language processor.

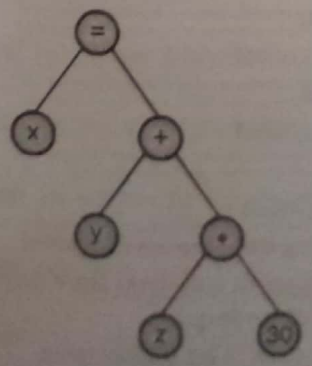
SPPU - May 2017, 3 Marks

In this phase, the compiler has to ascertain that each statement in the source file has been written as per the grammar.

- Tokens forming a statement are grouped together.
- Usually a source statement is represented using a parse tree. A syntax tree is a compressed representation of a parse tree.
- A syntax tree for the expression

  $x = y + z * 30$

  is shown in Fig. 4.2.3.
- In a syntax tree, operators appear as internal nodes and operand as leaf nodes.



(S4.5) **Fig. 4.2.3 : Syntax tree for x = y + z * 30**

Scanned with CamScanner

4.2.3 Semantic Analysis

(May 2017)

Q. Explain semantic analysis for language processor.

SPPU - May 2017, 3 Marks

Semantic analysis is for semantic errors. Type checking is an important aspect of semantic analysis. Each operator should have suitable operands. Some examples of semantic errors are given below.

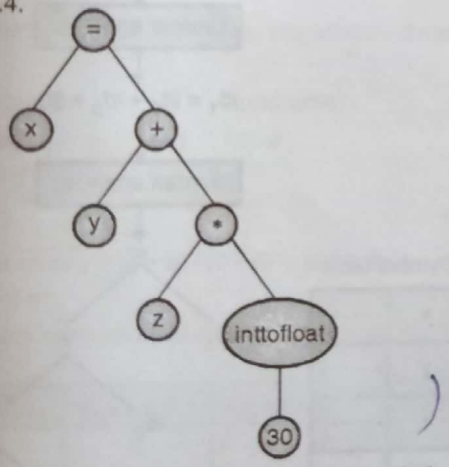
(1) 5* "ABC" [Multiplication of an integer and a string is not permitted].

(2) Multiplication of two pointers P1 and P2. P1 * P2 is not allowed.

(3) Many languages do not allow a real number as an index of an array.

In some cases, two operands can be made compatible through type casting.

- Many languages do not allow mixing of integer and real numbers in an expression.
  i.e. 5* 6.5 is not allowed by many languages.
- C-language permits mixing of real numbers and integer in an expression. C-language does type casting and upgrades the type of integer to real.
- In Fig. 4.2.3, if x, y and z are **float** type then the literal 30 should be converted into float. Let us assume the existence of an operator **inttofloat** that explicitly converts an integer into a float. Modified syntax tree after semantic analysis is shown in the Fig. 4.2.4.



(S4.6) **Fig. 4.2.4 : Syntax tree after semantic analysis**

Syllabus Topic : Compilation of Expressions

4.2.4 Intermediate Code Generation

Before generation of the machine code, the compiler creates an intermediate form of the source program. An intermediate code has the following properties :

(1) It should be easy to produce from the syntax tree.

(2) It should be easy to translate into machine code.

Intermediate code separates machine-independent phases (Lexical, Syntax, Semantic) of a compiler from the machine-dependent phases (code generation) of a compiler. The intermediate representation can have a variety of form. These forms include :

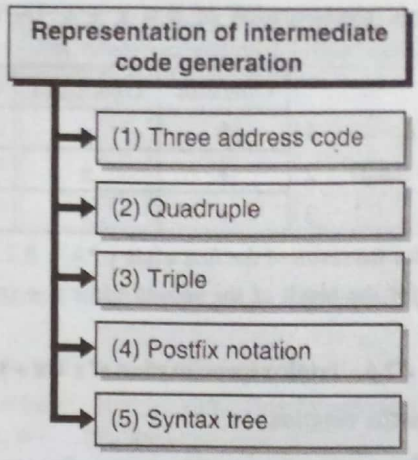


Fig. C4.3 : Representation of intermediate code generation

(1) Three address code

The three address code consists of a sequence of instructions, each instruction has maximum of three operands.

These instructions are similar to assembly instructions.

For example, to evaluate z = x + y *A

(1) y *A must be evaluated first and the result can be stored in a temporary variable temp1 (say).
temp1 = y * A

(2) temp1 must be added to x and the result can be stored in a temporary variable temp2 (say)
temp2 = x + temp1

(3) temp2 should be assigned to z.
z = temp2

Thus, the three address code corresponding to
z = x + y *A
is given by :

```
(1) temp1 = y *A
(2) temp2 = x + temp1
(3) z = temp2
```

(2) Quadruple representation

A quadruple representation has the following format :

Operator	Operand 1	Operand 2	Result

The quadruple representation of z = x + y *A is shown in the Fig. 4.2.5.

		Operator	Operand 1	Operand 2	Result
temp1 = y *A	1	*	y	A	temp1
temp2 = x + temp1	2	+	x	temp1	temp2
z = temp2	3	=	temp2	–	z

Fig. 4.2.5 : Quadruple representation of z = x + y *A

(3) Triple representation

A triple representation has the following format.

Operator	Operand 1	Operand 2

Scanned with CamScanner

The triple representation of z = x + y *A is shown in Fig. 4.2.6.

		Operator	Operand 1	Operand 2
temp1 = y *A	1	*	y	A
temp2 = x + temp1	2	+	x	(1)
z = temp2	3	=	z	(2)

'1' Stands for the result of the first triple y *A

'2' Stands for the result of the second triple x + (result of first triple).

Fig. 4.2.6 : Triple representation of z = x + y *A

(4) Postfix notation

In postfix notation, the operator appears after operands.

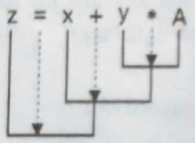
- A postfix notation is free from precedence.
- A postfix expression is simple to evaluate. Operators are evaluated in the order in which they appear in the expression.

The postfix representation of z = x + y *A is given by : z x y A* + =

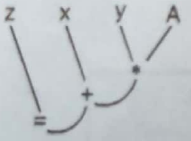
(5) Syntax tree

A syntax tree for an expression like z= x + y *A can be constructed using the following steps.

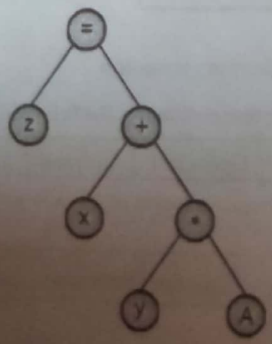
Step 1 : Grouping elements as per the sequence of evaluation.



Step 2 : Moving the operators at the centre of the groups.



Step 3 : Inverting the structure.



4.2.5 Code Optimization

In this phase, the compiler tries to improve the intermediate code so that a smaller and faster running machine code can be derived.

For example, the three instructions :

(1) temp1 = y *A

(2) temp2 = x + temp1

(3) z = temp2

can be written as,

(1) temp1 = y*A (2) z = x + temp1

Since, the temp2 is used only once, there is no point in storing the result of x + temp1 into temp2. It can be directly assigned to z.

4.2.6 Code Generation

It is easy to translate an intermediate instruction into a sequence of machine (assembly instruction). Translation of the above intermediate code is shown in Fig. 4.2.7.

Sr. No.	Intermediate code	Assembly code

1	temp1 = y*A	MOVER AREG, y
		MUL AREG, A
		MOVEM AREG, temp1

2	z = x + temp1	MOVER AREG, x
		ADD AREG, temp1
		MOVEM AREG, z

Fig. 4.2.7

Example 4.2.4

Explain the working of the various phases of the compiler for the expression :

x = y + z * 30

The variables x, y and z are of float type.

Solution :

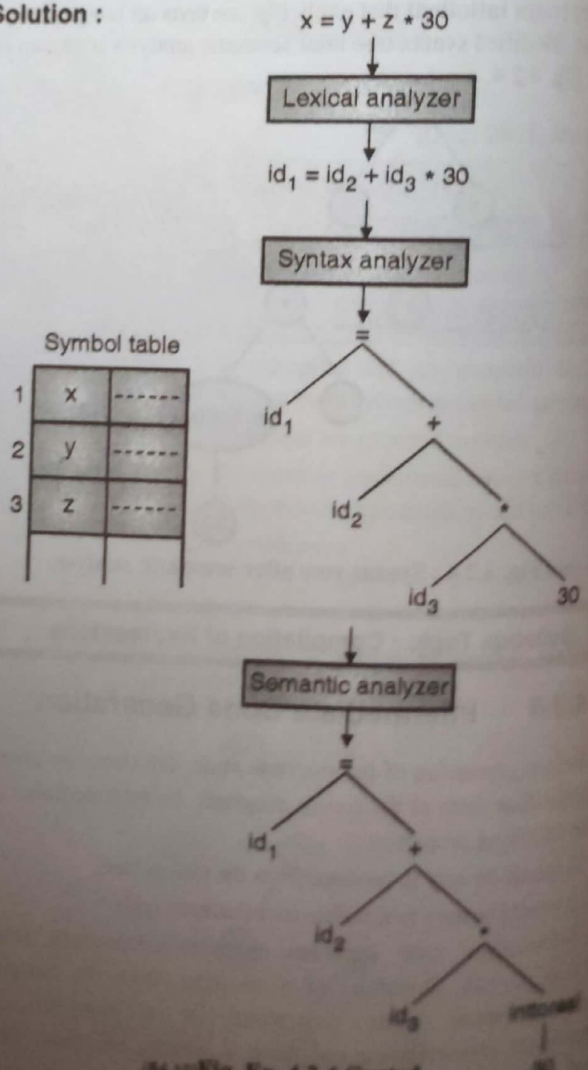
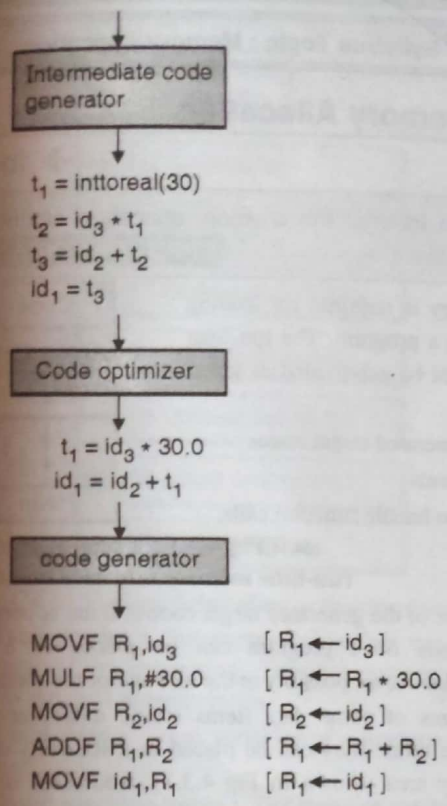


Fig. Ex. 4.2.4 Contd...

Scanned with CamScanner



(S4.10)**Fig. Ex. 4.2.4**

Example 4.2.5

For the statement given below, generate intermediate code in the format :

(i) Postfix notation (ii) Quadruple
(iii) Parse tree (iv) Triple

s = (a + b) / (c – d)

Solution :

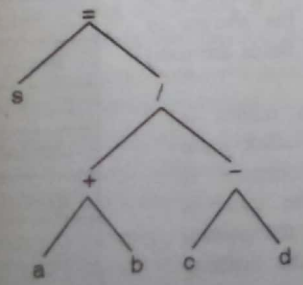
(i) Postfix notation of s = (a + b) / (c – d) is given by

s a b + c d – / =

(ii) The quadruple representation of s = (a + b) / (c –d) is given by :

	Operator	Operand 1	Operand 2	Result
1	+	a	b	t_1
2	–	c	d	t_2
3	/	t_1	t_2	t_3
4	=	t_3	–	s

(iii) The parse tree representation is given by



(S4.11)**Fig. Ex. 4.2.5**

(iv) Triple representation is given by

	Operator	Operand 1	Operand 2
1	+	a	b
2	–	c	d
3	/	(1)	(2)
4	=	s	(3)

Example 4.2.6

For the statement given below generate intermediate code in the format.

(i) Quadruple (ii) Triple
(iii) Parse tree (iv) Postfix notation.

A = – P * (– Q + R)

Solution :

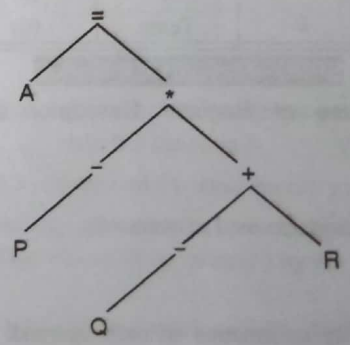
(i) Quadruple

	Operator	Operand 1	Operand 2	Result
1	–	P	–	t_1
2	–	Q	–	t_2
3	+	t_2	R	t_3
4	*	t_1	t_3	t_4
5	=	t_4	–	A

(ii) Triple

	Operator	Operand 1	Operand 2
1	–	P	–
2	–	Q	–
3	+	(2)	R
4	*	(1)	(3)
5	=	A	(4)

(iii) Parse tree



(S4.12)**Fig. Ex. 4.2.6**

(iv) Postfix notation

AP – Q – R + * =

Example 4.2.7

What are different intermediate code forms, used in compilers ? Generate intermediate codes, of at least 4 forms, for the assignment statement given below :

Temp = limit * (max – min) + 3 * limit * (max + min)

Solution : Several forms are used for representation of intermediate code. These forms include :

(1) Postfix notation (2) Quadruple
(3) Parse tree (4) Triple

Scanned with CamScanner

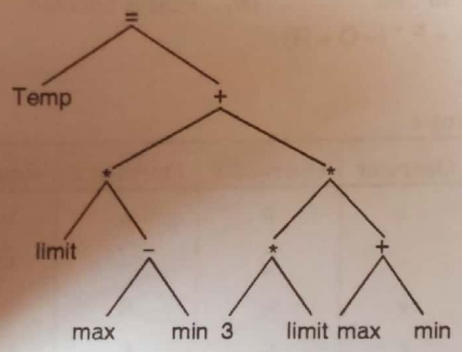
(1) Postfix representation

Temp limit max min – * 3 limit * max min + * + =

(2) Quadruple representation

Operator	Operand 1	Operand 2	Result
–	Max	Min	t_1
*	Limit	t_1	t_2
*	3	Limit	t_3
+	Max	Min	t_4
*	t_3	t_4	t_5
=	t_2	t_5	Temp

(3) Parse tree representation



(S4.13)**Fig. Ex. 4.2.7**

(4) Triple representation:

	Operator	Operand 1	Operand 2
1	–	Max	Min
2	*	Limit	(1)
3	*	3	Limit
4	+	Max	Min
5	*	(3)	(4)
6	+	(2)	(5)
7	=	Temp	(6)

Example 4.2.8 **SPPU - May 2013, 6 Marks**

Explain the use of Register Descriptor and Operand descriptor.

Solution :

The operand descriptor is used to maintain

1. Type
2. Length
3. Addressability information of each operand. Durning code generation phase, the complier takes a decision about the machine instruction to be used by analysing an operator and the descriptors of its operands.

– An operand descriptor is built for every operand in an expression.

– An operand descriptor is used during code generation phase.

– The fields of an operand descriptor are its attributes and its addressability.

– Similarly, a register descriptor has two fields.

1. **Status :** indicating free or occupied
2. **Operand descriptor :** if occupied then the operand descriptor of the value in the stored register.

Syllabus Topic : Memory Allocation

4.3 Memory Allocation

→ (Dec. 2015)

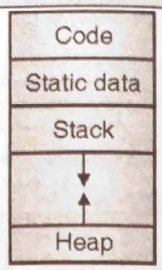
Q. Explain in brief the memory allocation algorithms with examples. **SPPU - Dec. 2015, 6 Marks**

Memory is required for loading and running a program. The run-time storage might be subdivided as given below :

(1) The generated target code.

(2) Data items

(3) Stack to handle function calls.



(S4.14)**Fig. 4.3.1 : A typical subdivision of run-time memory into code and data areas**

– The size of the generated target code is fixed at compile time. The code of a program can be placed in a statically determined area, possibly at the low end of the memory.

– The sizes of some data items (static data) are known at compile time. They can be placed in a statically determined memory area (shown in Fig 4.3.1). Addresses of statically allocated data items can be compiled into target code.

– An extended stack is used to manage function calls. In case of a function call :

(1) The status of the machine, such as the value of the program counter and machine registers is saved on the stack.

(2) When the control returns from the called function, the original state of the calling program can be restored from the stack.

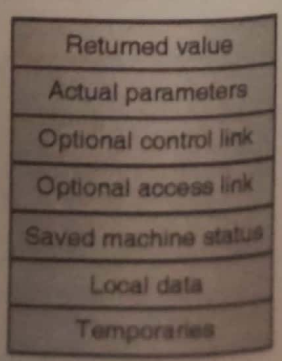
(3) Data items of the called function can be allocated on the stack along with other information associated with function call.

– Many programming languages like Pascal require a separate memory area, called a heap, to hold other information. Implementation of language in which lifetimes of variables cannot be handled through stack will use the heap.

Activation records

Information need to call and execute a function is managed through an activation record. An activation record uses a contiguous block of storage. It consists of a number of fields (as shown in Fig. 4.3.2). Not all fields are used by all compilers.

– An activation record is pushed on the stack when a function/procedure is called and the same is popped when the control returns to the calling program.



(S4.15)**Fig. 4.3.2 : A general activation record**

Scanned with CamScanner

The various fields of an activation record are being explained below :

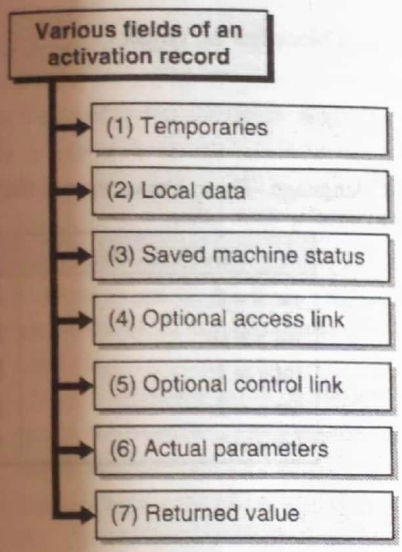


Fig. C4.4 : Various fields of an activation record

- **(1) Temporaries :** Temporary values are needed during evaluation of an expression. These temporary values are stored in the field of temporaries.
- **(2) Local data :** Local variables of a function /procedure are stored here.
- **(3) Saved machine status :** The status of machine is saved before transferring control to the called function. This information includes the value of program counter and CPU registers. This information must be restored when the control returns from the called function.
- **(4) Optional access link :** Optional access links are used to access non-local data in block-structured languages. This link is not required for a languages like Fortran where the nonlocal data is kept in a fixed place.
- **(5) Optional control link :** The optional control link points to the activation record of the caller.
- **(6) Actual parameters :** This field is used by the calling function to pass parameters to the called function.
- **(7) Returned value :** This field is used by the called function to return a value to the calling function.

4.3.1 Static and Dynamic Memory Allocation

(Oct. 2016)

Q. Explain static and dynamic memory allocation.

SPPU - Oct. 2016 (In Sem), 4 Marks

In static allocation of memory, the size of the generated target code and the size of data items are known at compile time. They can be placed in a statically determined memory area.

- In static memory allocation, memory is allocated to variables before the execution of a program begins.
- In static memory allocation, allocation is performed during compilation. No memory allocation or de-allocation is performed during program execution.
- In static memory allocation, variables (both local and nonlocal) remain allocated. Allocation to variables exists even if the function is not active.

Disadvantages of static memory allocation

(1) It is almost impossible to handle recursion.

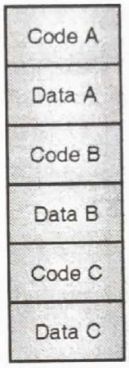
(2) Memory requirement is higher. Variables remain allocated even when the function is not active.

Advantages of static memory allocation

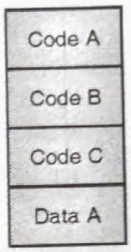
(1) It is simple to implement. Binding is performed during compilation.

(2) Execution is faster as there is no binding during the runtime and addresses of variables is directly encoded in machine instructions.

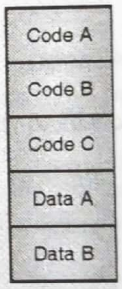
In dynamic memory allocation, memory is allocated and de-allocated during execution of a program. Here, memory bindings are established and destroyed during execution of a program.



(a) Static allocation



(b) Dynamic allocation. The main program A is active



(c) Dynamic allocation. The main program A calls the function B

(S4.16) **Fig. 4.3.3 : Static and dynamic memory allocation**

- Dynamic memory allocation is characterized by automatic allocation. This allocation is controlled by the program under execution.
- Memory is allocated to variables declared inside a function when the function is called. This memory is de-allocated when the control returns back to the calling program.
- Dynamic memory allocation has optimal utilization of memory. The same memory can be used for variables of different functions/procedures.
- Dynamic memory allocation is implemented using stacks. It requires pointer based access to variables. Pointer based accesses are slower. Individual variables stored in an activation record can be accessed by adding :
  (1) Starting address of the activation record.
  (2) Offset of the variable inside the activation record.
- Recursion can be implemented easily using dynamic memory allocation.

Scanned with CamScanner

4.3.2 Memory Allocation in Block Structured Languages

A block is a list of statements and it can contain its own local data declarations. In C, a block has the syntax

```
{
        declarations
        statements
}
```

A block can have nested structure. A block is confined inside braces {and} in C-language. These braces ensure that one block is either independent of another, or is nested inside the other. Two blocks are not allowed to overlap each other.

```
main()
{
        int x = 0;
        int y =0;
        {
                int
                {
                y = 1;

                int x = 2;
                printf ("%d, %d", x, y);
                }
        B0      B1      B2

                int y = 3;
                printf("%d, %d", x, y);
                }
                B3
                }
                printf("%d %d", x, y);

        printf("%d %d", x, y);
}
```

Declaration	Scope
int x = 0;	$B_0 - B_2$
int y = 0;	$B_0 - B_1$
int y = 1;	$B_1 - B_3$
int x = 2;	B_2
int y = 3;	B_3

Fig. 4.3.4 : Blocks in a C-Program

The scope of a variable declaration in a block-structured language is given by the following nesting rules :

(1) The scope of a variable declared inside a block B includes B.

(2) If a variable x is not declared in a block B_1, then usage of x in B_1 is in the scope of a declaration of x in an enclosing block B such that :

(i) B has a declaration of x, and

(ii) B is more closely nested around B_1 than any other block with a declaration of x.

The Fig. 4.3.4 is a sample C-Program with nesting of blocks. Variables x and y are initialized to block numbers in which they are declared.

- The scope of declaration y in B_0 does not include B_1 as y is redeclared in B_1, it is indicated by B_0–B_1.
- The values of x and y in these blocks is given by :

x	y	Block in which the values are printed
2	1	B_2
0	3	B_3
0	1	B_1
0	0	B_0

Implementation of block structure using a stack

Block structure can be implemented using a stack. Block can be treated as a parameterless function/procedure. Space for variables is allocated on the stack when the block is entered, and de-allocated when control leaves the block.

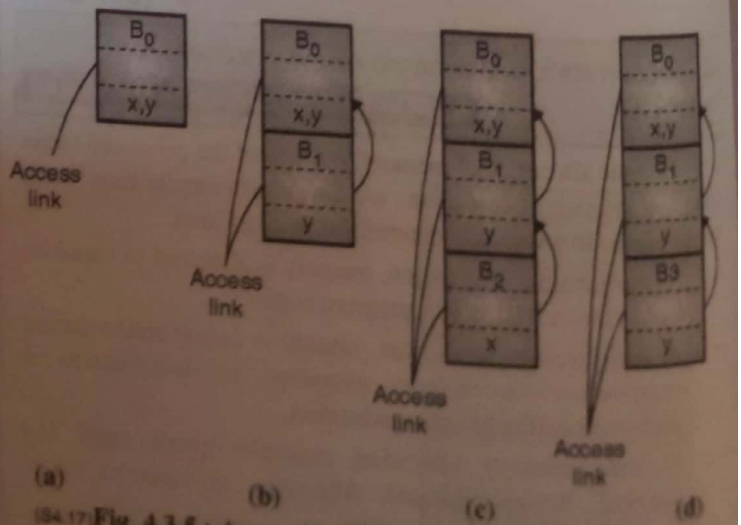


Fig. 4.3.5 : Access links for finding non-local variables

Scanned with CamScanner

- For finding the scope of variables, an access link is added to each activation record.
- If a block B_1 is nested inside the block B_0, then the access link in the activation record for B_1 points to the access link in the record for the most recent activation of B_0.
- Fig. 4.3.5 is a snapshot of the run-time stack during an execution of the program in Fig. 4.3.4.

Fig. 4.3.5 (a) : Block B_0 is entered. Two variables x and y are declared inside the block B_0.

Fig. 4.3.5 (b) : Control enters the block B_1. Block B_1 is nested inside the block B_0. The variable y is redeclared inside B_1.

Fig. 4.3.5 (c) : Control enters the block B_2. Block B_2 is nested inside the block B_1. The variable x is redeclared inside B_2.

Fig. 4.3.5 (d) : Control leaves the block B_2 and enters the block B_3. The block B_2 is deallocated and block B_3 is allocated on the stack. The variable y is redeclared inside B_3.

Static Implementation of block structure using a stack

Storage for a complete function/procedure can be made at one time. Necessary actions can be taken for variable declaration inside the blocks. For the entire block B_0 (Fig. 4.3.4), we can make memory allocation as shown in Fig. 4.3.6.

(S4.18) **Fig. 4.3.6 : Storage for variable declared in Fig. 4.3.4**

The subscripts on local variables x and y identify the blocks in which these local variables are declared. It may be noted x_2 and y_3 can be assigned the same storage because they are in blocks that are not alive at the same time.

4.3.3 Array Allocation and Access

Elements of an array are stored in consecutive locations. Let us consider an array

a [low .. high]

index varies from low to high. Number of elements in the array is given by

high – low + 1

If the width of each array element is w, then the address of a[i] is given by :

base + (i – low) × w

where base is the starting address of the storage allocated to the array.

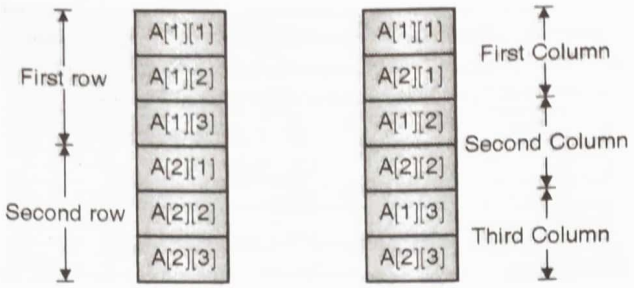
In case of a two dimensional array stored in row-major form, the address of $A[i_1][i_2]$ in the array $A[low_1 - high_1][low_2 - high_2]$ can be calculated by the formula.

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

where n_2 is the number of columns. n_2 is given by

$$n_2 = high_2 - low_2 + 1$$

A layout of two-dimensional array is shown in Fig. 4.3.7.



(a) Row-major (b) Column-major

(S4.19) **Fig. 4.3.7 : Layouts for a two-dimensional array**

Example 4.3.1 **SPPU - May 2014, 8 Marks**

Explain parameter passing mechanisms :

(i) call by value (ii) call by result

(iii) call by references (iv) call by name.

Solution :

(i) Call by value

In this method, the values of actual parameters are passed to formal parameters (parameters defined in called function). Any changes to the values of formal parameters are not reflected in actual parameters (defined in calling program).

(ii) Call by result

It is similar to call by value with one difference. At the time of return from the called function, the values of formal parameters are copied back into corresponding actual parameters.

(iii) Call by reference

In this method, the address of an actual parameter is passed to the called function. The called function can access and modify the values of actual parameters.

(iv) Call by name

In this method, every occurrence of a formal parameter is the body of the called function is replaced by the name of the corresponding actual parameter.

Syllabus Topic : Compilation of Control Structures

4.4 Compilation of Control Structure

There are language features which govern the sequencing of control through a program. Some of the common control structures are :

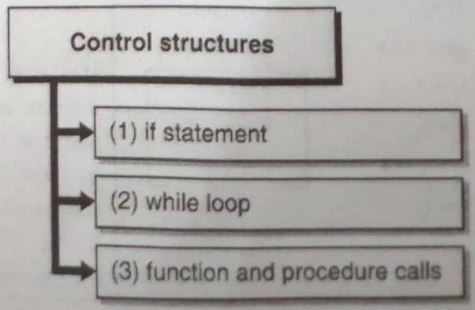


Fig. C4.5 : Control structures

Scanned with CamScanner

4.4.1 if Statement

We can translate an if-statement into three-address code. A program containing if-statement can be mapped into an equivalent program containing explicit goto's. A sample if statement is compiled into three-address code in Fig. 4.4.1.

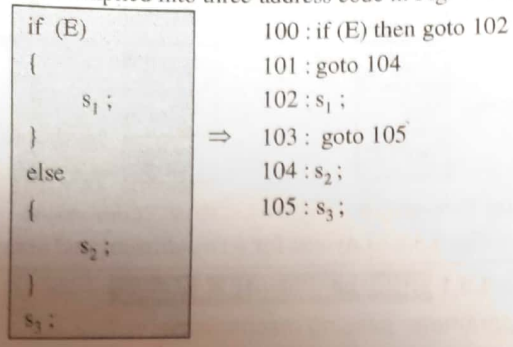


Fig. 4.4.1 : Translation of if-statement

- It should be clear that if E is true then the two statements S_1 and S_3 will be executed.
- If E is false then the two statements S_2 and S_3 will be executed.

Example 4.4.1

Generate intermediate code for the following program segment.

```
if (a < b || c < d && e < f)
    {
        x = x + 1 ;
        y = y – 1 ;
    }
else
    {
        x = x – 1 ;
        y = y + 1 ;
    }
```

Solution

```
100 :   if a < b goto 105
101 :   if c < d goto 103
102 :   goto 110
103 :   if e < f goto 105
104 :   goto 110
105 :   t1 = x + 1
106 :   x = t1
107 :   t2 = y – 1
108 :   y = t2
109 :   goto 114
110 :   t1 = x – 1
111 :   x = t1
112 :   t2 = y + 1
113 :   y = t2
114 :
```

Execute
x = x + 1;
y = y – 1;

If a < b then a < b || c < d && e < f is true.

execute
x = x – 1;
y = y + 1;

If a < b is false then both c < d and e < f should be true for the statement a < b || c < d && e < f to be true.

4.4.2 while Statement

We can translate a while-statement into three-address code. A program containing while-statements can be mapped into an equivalent program containing explicit goto's. A sample while statement is compiled into three-address code in Fig. 4.4.2.

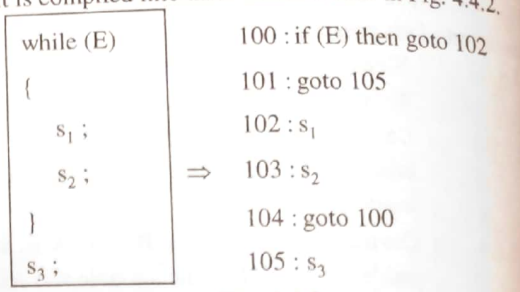


Fig. 4.4.2

- It should be clear that if E is true then control enters the body of the loop and executes s_1 and s_2. Subsequently, it goes back to the beginning of the loop.
- If E is false then control is transferred to the statement s_3.

Example 4.4.2

Generate the quadruples for the code given below.

```
for (j = 0; j < = 5; j ++)
    x [2 * j] = y [2 * j + 5]
```

where x and y are single dimensional arrays with lower and upper bounds as 0 and 5. Also generate the quadruples for the same after applying machine code optimization of any type.

Solution :

Base address of the array x is being assumed as base x. The base address of the array y is assumed as base y. Word size is taken as w.

The code can be optimized through elimination of the common sub-expression 2*j.

The three address code is given below :

(1)	$j = 0$	(2)	$t_1 = j > 5$
(3)	if t_1 then goto 13	(4)	$t_2 = 2 * j$
(5)	$t_3 = t_2 * w$	(6)	$t_4 = \text{base } x + t_3$
(7)	$t_5 = t_2 + 5$	(8)	$t_6 = t_5 * w$
(9)	$t_7 = \text{base } y + t_6$	(10)	$* t_4 = * t_7$
(11)	$j = j + 1$	(12)	goto 2
(13)			

Quadruples

	Operator	Operand 1	Operand 2	Result
1	=	0		j
2	>	j	5	t_1
3	if	t_1		13
4	*	2	J	t_2
5	*	t_2	w	t_3
6	+	base x	t_3	t_4
7	+	t_2	5	t_5
8	*	t_5	w	t_6

Scanned with CamScanner

	Operator	Operand 1	Operand 2	Result
9	+	base y	t_6	t_7
10	=	* t_7		* t_4
11	+	j	1	j
12	goto			2

Syllabus Topic : Code of Optimization

➜ 4.4.3 function and procedure call

Function call involves passing a parameter and also the return address must be saved on the stack. This concept is explained in section 4.3.2.

4.5 Code Optimization

➜ (May 2013, Oct. 2016, Dec. 2016)

Q. List down various code optimization techniques. Explain any two techniques in detail with example.

SPPU - May 2013, Oct. 2016(In Sem), 8 Marks

Q. What is the need for code optimization ? Explain various code optimization techniques.

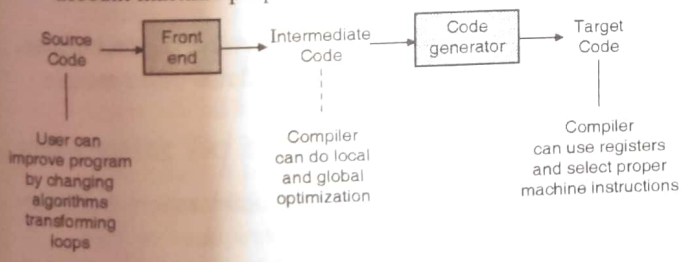
SPPU - Dec. 2016, 7 Marks

Code optimization is technique of producing code that can run faster or take less space, or both. This is achieved in the following ways :

(1) Elimination of redundancies.

(2) Computations in program are rearranged or rewritten to make it run faster.

- Code optimization has nothing to do with program algorithm.
- Code optimization that we will discuss will not take into account machine properties like instruction set, registers etc.



(S4.20) **Fig. 4.5.1 : Places for improvements by the user and the compiler**

- Improvement in running time can be obtained by improving a program at various levels (Fig. 4.5.1).
- Compiler can replace a sequence of operations by an algebraic sequence and thereby reduce the running time of a program.

A compiler can carry out several transformations of a program at intermediate code level without changing its meaning. A few techniques used in compilers are given below :

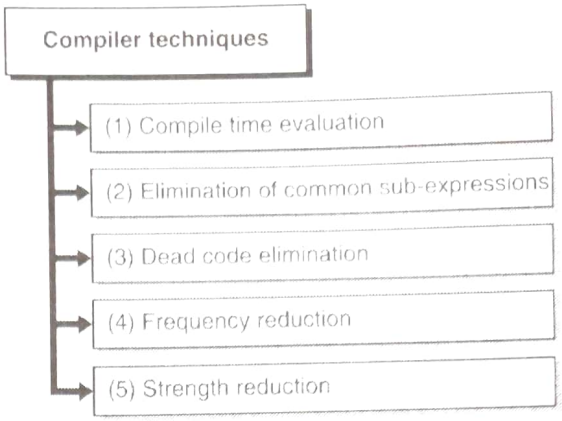


Fig. C4.6 : Compiler techniques

➜ (1) Compile time evaluation

Certain operations can be performed during compile time. When all operands in an operation are constants, the operation can be performed during compilation.

☞ **Example**

$x = 5 * 3 + 9 + y$

can be written as

$x = 24 + y$

This will eliminate multiplication and addition operations during execution of the program. Thus, the execution time of a program can be reduced by compile time evaluation of certain operations.

➜ (2) Elimination of common sub-expressions

Many instructions may contain common sub-expressions. We can avoid re-computing a common sub-expressions by using its previously computed value. An occurrence of an expression E is called a common sub-expressions if E was previously computed, and the values of variables in E have not changed since previous computation.

☞ **Example**

Let us consider a program segment

```
x = y * z ;
=======
a = y * z + 6 ;
```

The sub-expressions 'y*z' is common in both the expressions. The value of the sub-expressions 'y*z' can be saved in a temporary variable t_1. Now, t_1 can be used in place of 'y*z'.

```
x = y * z;
=======
a = y * z + 6 ;
Original program
```

⇒

```
t1 = y * z;
x = t1 ;
=======
a = t + 6 ;
After optimization
```

➜ (3) Dead code elimination

A program may contain useless code. Useless code compute values that never get used. Dead code is detected by checking whether the value assigned to a variable is used subsequently in the program.

Scanned with CamScanner

☞ **Example**

Let us consider a program segment.

```
a = 5;                    a = 5;
b = 3;                    b = 3;
x = a * b + 6;    ⇒       Print a, b;
Print a, b;
```

The value assigned to x is not used subsequently. Therefore, the statement 'x = a * b + b ;' can be eliminated.

➜ **(4) Frequency reduction**

Some instructions can be moved out of a loop. If there is an expression that yields the same result independent of the number of times a loop is executed. Such expressions can be placed before the loop.

☞ **Example**

```
while (i < = 100)          z = 5 ;
{                          while (i < = 100)
    z = 5 ;                {
    x = z * i ;      ⇒         x = z * i ;
    i ++ ;                     i ++ ;
}                          }
```

The statement 'z = 5;' is a loop-invariant computation. The meaning of the code will not change even if the expression 'z = 5;' is placed before the loop. Moving a statement outside the body the loop reduces its frequency of execution.

➜ **(5) Strength reduction**

Strength reduction involves replacement of a time consuming operation (a high strength instruction) by a faster operation (a low strength instruction). In some cases the multiplication operation can be replaced by an addition.

☞ **Example**

x = 2*y can be written as x = y + y

x^2 can be written as x * x.

4.6 Exam Pack (University Questions)

☞ **Syllabus Topic : Basic Compliers Function**

Q. Explain : Complier.
(Refer section 4.1)(4 Marks)**(Dec. 2013, Dec. 2014)**

Q. Explain synthesis phase of a compiler.
(Refer section 4.1(2)) (6 Marks) **(Dec. 201**

☞ **Syllabus Topic : Phases of Compilation**

Q. Explain phases of a compiler with example.
(Refer section 4.2) (6 Marks)
(Aug. 2015(In Sem), May 201

Q. Explain lexical analysis with example.
(Refer section 4.2.1) (3 Marks) **(Dec. 201**

Q. Explain lexical analysis for language processo
(Refer section 4.2.1) (3 Marks) **(May 201**

Q. Explain syntax analysis with example.
(Refer section 4.2.2) (3 Marks) **(Dec. 201**

Q. Explain syntax analysis for language processo
(Refer section 4.2.2) (3 Marks) **(May 201**

Q. Explain semantic analysis for language processo
(Refer section 4.2.3) (3 Marks) **(May 201**

☞ **Syllabus Topic : Compilation of Expressions**

Example 4.2.8 (6 Marks) **(May 201**

☞ **Syllabus Topic : Memory Allocation**

Q. Explain in brief the memory allocation algorithms with examples.
(Refer section 4.3) (6 Marks) **(Dec. 201**

Q. Explain static and dynamic memory allocation.
(Refer section 4.3.1) (4 Marks)
(Oct. 2016 (In Sem)

Example 4.3.1 (8 Marks) **(May 201**

☞ **Syllabus Topic : Code of Optimization**

Q. List down various code optimization techniques. Explain any two techniques in detail with example.
(Refer section 4.5) (8 Marks)
(May 2013, Oct. 2016(In Sem))

Q. What is the need for code optimization ? Explain various code optimization techniques.
(Refer section 4.5) (7 Marks) **(Dec. 2016)**

□□□

Scanned with CamScanner

UNIT II

CHAPTER 5

Loaders

Syllabus Topics

Loader Schemes : Compile and go, General Loaders Scheme, Absolute loaders, subroutine linkages, relocating loaders, direct linking loaders, Design of absolute loader.

5.1 Basic Functions of a Loader

➔ (May 2014, Dec. 2014, Aug. 2015, Dec. 2015)

Q. What is loader ? List basic functions and features of a loader. **SPPU - May 2014, Dec. 2015, 8 Marks**

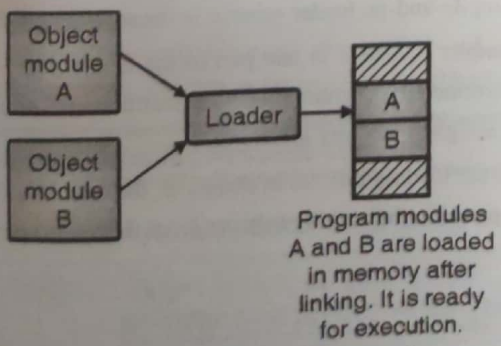
Q. Explain the terms : Loader, Linker. **SPPU - Dec. 2014, 4 Marks**

Q. Explain functions of a Loader. **SPPU - Aug. 2015(In Sem), 4 Marks**

A source program is converted to object program by assemblers and compilers. The loader is a program which accepts object codes and prepares them for execution and initiates execution. As many as four functions are performed by a loader. These functions are :

1. **Allocation** of space in main memory for the programs.
2. **Linking** of object modules with each other. Linking involves resolving of symbolic references between object modules.
3. Adjust all address dependent locations, such as address constants, to correspond to the allocated space. It is also called **relocation.**
4. Physically **loading** the machine instructions and data into the main memory.

A general loading scheme is shown in the Fig. 5.1.1



(S5.1)**Fig. 5.1.1 : General loading scheme**

☞ **What is linking ?** ➔ **(May 2014)**

Q. What is linker ? **SPPU - May 2014, 4 Marks**

Any usable program written in any language has to use functions/ subroutines. These functions could be either user defined functions or they can be library functions.

For example, consider a program written in C-language. Such a program may contain calls to functions like printf() and scanf(). During program execution the main program as well as functions must reside in the main memory. In addition, every time a function is called, the control should get transferred to the appropriate function.

- The linking process makes address of modules known to each other so that transfer of control takes place during execution.
- Passing of parameters is handled by the linker. Parameters can be passed by value or by reference. A value returned by a function must be handled.
- Every public variable should have the same address in every module. An external variable can be defined in one module and can be used in another module. The address of an external variable should be same in every module. Resolving of addresses of symbolic references is handled by the linker.

☞ **What is relocation ?** ➔ **(May 2013)**

Q. Why program relocation is required and how is it performed? **SPPU - May 2013, 6 Marks**

Relocation is the process of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from any designated area of memory.

The statement

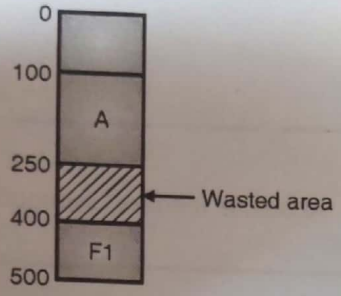
MOVER AREG, X

This is an address sensitive instruction. For this instruction to execute correctly, the actual address of X should be put in the instruction.

Scanned with CamScanner

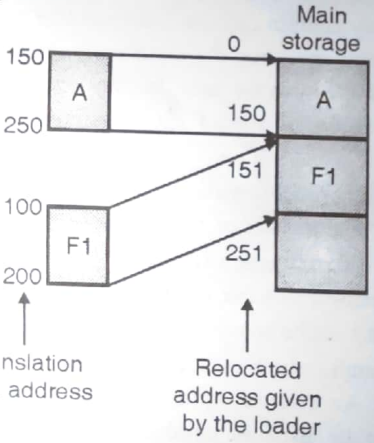
Assume that a program written in C-language (let us call it A) calls a function F1. The program A and the function F1 must be linked with each other. But when in main storage shall we load A and F1 ? A possible solution would be to load them according to the addresses assigned when they were translated.

Case I : At the time of translation, A has been given storage area from 100 to 250 while F1 occupies area between 400 to 500. If we were to load these programs at their translated addresses, a lot of storage lying between them will be wasted.



(S5.2)**Fig. 5.1.2 : Case I of relocation**

Case II : At the time of translation, both A and f_1 may have been translated with the identical start address 100. A goes from 100 to 250 and f_1 goes from 100 to 200. These two modules cannot co-exist at same storage locations. The loader must relocate A and F1 to avoid address conflict or storage waste A possible relocation is shown in Fig. 5.1.3.



(S5.3)**Fig. 5.1.3 : Relocation to avoid address conflict or storage waste**

- It may be noted that relocation is more than simply moving a program from one area to another in the storage. It refers to adjustment of address fields and not to movement of a program.

Example 5.1.1 SPPU - Dec. 2014, Oct. 2016(In Sem), May 2017

Explain Translated origin, Link origin, Load origin.

Solution :

- **Translated origin :** It is the address specified by the programmer in an ORIGIN statement. This address is used by the translator for code generator.
- **Linked origin :** This address of the origin assigned by the linker while producing the machine code.
- **Load origin** : Address of the origin assigned by the loader while loading the program for execution.

Syllabus Topic : Loader Schemes

5.2 Loading Schemes

→ (May 2014, Dec. 2014, Dec. 2015,Dec. 2016)

Q. List and explain the different loader scheme in brief.

SPPU - May 2014, Dec. 2014, Dec. 2015, Dec. 2016, 8 Marks

There are several schemes for accomplishing the four loading functions. These schemes are :

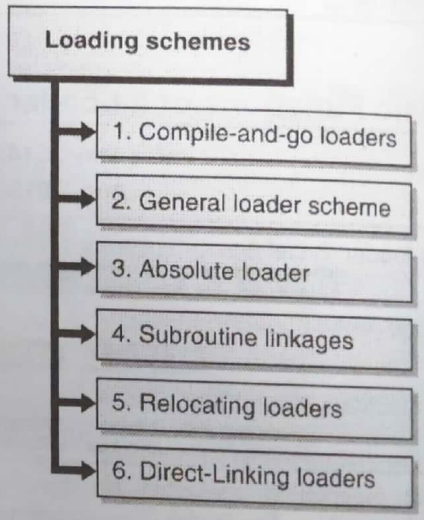


Fig. C5.1 : Loading schemes

Syllabus Topic : Compile and go

→ 5.2.1 Compile-and-go Loaders

→ (Dec. 2014, Dec. 2015, Oct. 2016)

Q. Explain any one type of loader.

SPPU - Dec. 2014, 3 Marks

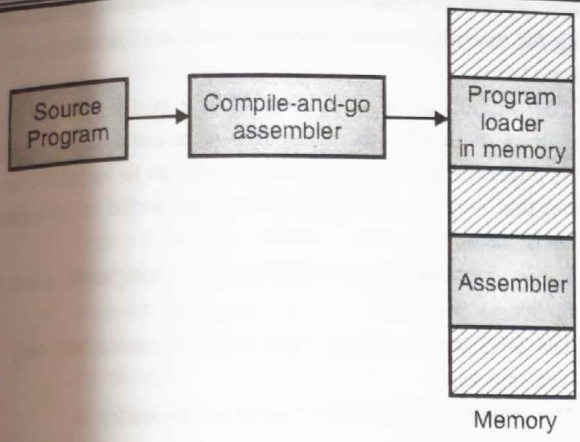
Q. Explain compile and go-loader scheme.

SPPU - Dec. 2015, Oct. 2016(In Sem), 6 Marks

A compile-and-go loader scheme is shown in Fig. 5.2.1.

- Assembler is loaded in one part of the memory and it places the assembled program (machine instructions) directly into their assigned memory locations.
- After the loading process is complete, the assembler transfers the control to the starting instruction of the loaded program.

Scanned with CamScanner



(S4.21)**Fig. 5.2.1 : Compile-and-go loader scheme**

☞ Advantages of compile and-go-loader

(i) It is easy to implement.

(ii) It is a very simple scheme.

☞ Disadvantages of compile and-go-loader

(i) A portion of the memory is wasted as it is occupied by the assembler.

(ii) It is required to re-translate the user program every time it is run.

(iii) It is difficult to handle multiple segments.

Thus, it becomes very difficult to develop a modular program under compile-and-go assembler.

Syllabus Topic : General Loader Scheme

➔ 5.2.2 General Loader Scheme

A general loading scheme is shown in Fig. 5.1.1.

- Output of assembler is saved in a file. This output can be loaded and executed whenever the user wants to run the program. The output of the assembler is known as the object program.
- With the use of output of assembler as intermediate file, the disadvantages of "compile-and-go" scheme can be removed :
  1. Retranslation of program every time it is run is avoided.
  2. The assembler need not reside in the main memory at the time of execution. Only the loader need to be in the memory but it is much smaller compare to a compile-and-go loader.
  3. It is possible to write a modular program under general loading scheme.
- In this scheme, the loader accepts object files and places machine instructions and data in the memory for execution.

Syllabus Topic : Absolute Loaders

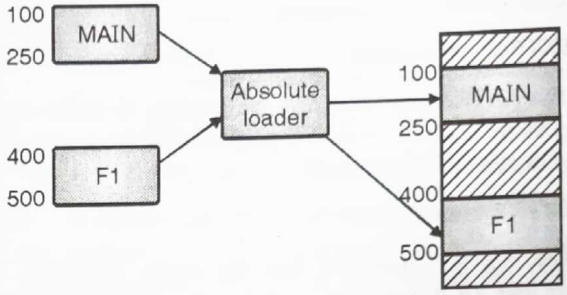
➔ 5.2.3 Absolute Loader ➔ (Aug. 2015)

Q. Explain Absolute Loaders loading scheme.

SPPU - Aug. 2015(In Sem), 6 Marks

The task of an absolute loader is virtually trivial. The loader simply accepts the machine language code produced by an assembler and places it into main memory at the location specified by the assembler.

Fig. 5.2.2 illustrates the operation of an absolute loader. The main program is assigned to locations 100 to 250 and the function F1 is assigned locations 400 to 500. The programmer should be careful not to assign overlapping locations to modules to be linked.



(S5.4)**Fig. 5.2.2 : Absolute loader example**

The four loading functions are accomplished in an absolute loading scheme :

1. Allocation – by programmer
2. Linking – by programmer
3. Relocation – by assembler
4. Loading – by loader

☞ Advantages of absolute loading scheme

1. No relocation information is required, so the size of the object module is comparatively small.
2. This scheme is very simple to implement.
3. This scheme makes more memory available for loading since the assembler is not in memory at the time of loading.
4. No modification of address sensitive entities is required at the time of loading.
5. This scheme supports multiple object modules to reside in memory.

☞ Disadvantages of absolute loader

1. Since the linking is handled by the programmer, programmer has to remember the address of each module and use that absolute address explicitly for linking.
2. The programmer has to be careful not to assign overlapping locations to modules to be linked.
3. Lot of memory lying between modules will be wasted.
4. If changes are made to one module that increases it size then it can overlap the start of another module. It may require manual shifting of module. This manual shifting can become very complex and tedious.

Syllabus Topic : Subroutine Linkages

➔ 5.2.4 Subroutine Linkage ➔ (Dec. 2013)

Q. Explain sub routine linkers.

SPPU - Dec. 2013, 4 Marks

Scanned with CamScanner

A program consisting of main program and a set of functions (subroutines) could reside in several files. These program units are assembled separately.

The problem of subroutine linkage is this : a main program A wishes to call the subroutine B and if the subroutine B resides in another file then the assembler will not know the address of B and declare it as an undefined symbol.

To realize such interactions, A and B must contain public definitions and external references.

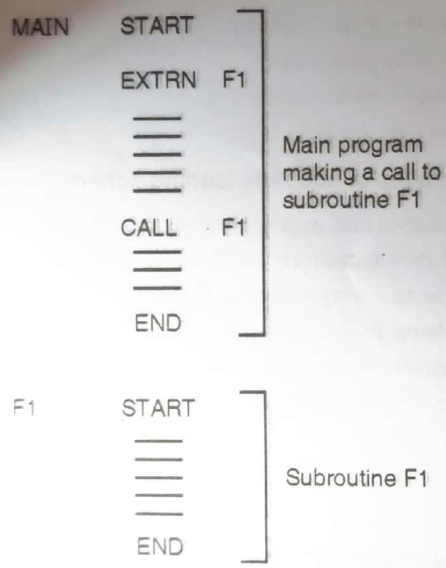
☞ EXTRN statements

The EXTRN statement lists the symbols to which external references are made in the current program unit. These symbols are defined in other program units.

☞ ENTRY statements

The ENTRY statement lists the public definitions of a program unit, i.e. it lists those symbols defined in the program unit which may be referenced in other program units.

For example, the following sequence of instruction may be a simple calling sequence to another program :



(S5.5)**Fig. 5.2.3**

- In this example, the subroutine F1 is declared as an external symbol in the main program.
- Assembles can not provide the addresses of external symbols. Their addresses are fixed at the time of linking. Thus, an external reference is said to be unresolved until linking is performed for it. It is said to be resolved when its linking is completed.

Syllabus Topic : Relocating Loaders

→ 5.2.5 Relocating Loader → (Dec. 2013)

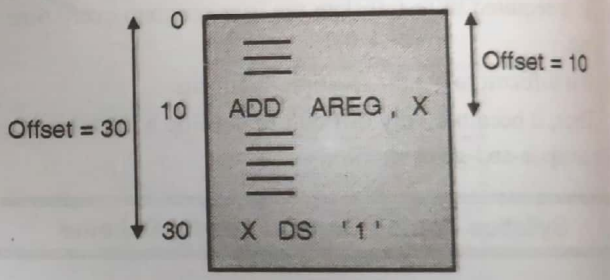
Q. Explain relocation loaders.

SPPU - Dec. 2013, 4 Marks

To avoid possible assembling of all subroutines when a single subroutine is changed and to perform the tasks of allocation and linking for the programmer, the general class of relocating loader was introduced.

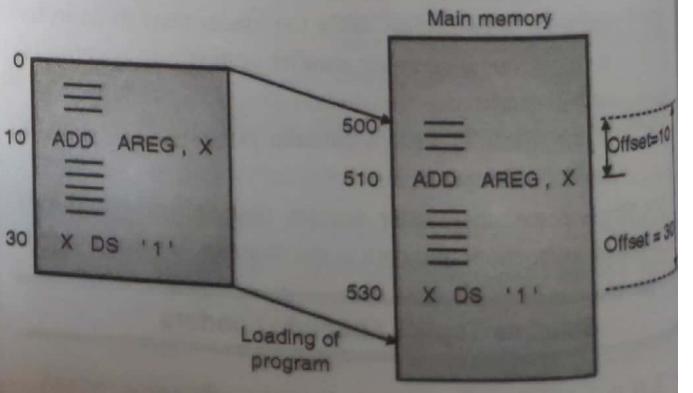
The output of a relocating loader is the program and information about all other programs it references. In addition there is information (relocation information) as to locations in this program that must be changed if it is to be loaded in an arbitrary location in memory.

- Binary symbolic loader (BSS) is an example of relocating loader.
- The BSS loader allows many code segments but only one data segment.
- The output of the assembler using a BSS loader is :
  1. Object program
  2. Reference about other programs to be accessed.
  3. Information about address sensitive entities.
- The relocation requirements of a program are influenced by the addressing structure of the computer system on which it is to execute. Many computer systems provide hardware support for relocation. Use of segment registers reduces the relocation requirement of a program. Let us consider a program segment as shown in Fig. 5.2.4.



(S5.6)**Fig. 5.2.4 : A sample program segment considered for relocation**

- In the above program segment the address of variable X in the instruction
  ADD AREG, X
  will be 30
- If this program is loaded from the memory location 500 for execution them the address of X in the instruction
  ADD AREG, X
  must become 530, it is shown in Fig. 5.2.5.



(S5.7)**Fig. 5.2.5 : Relocation of the program**

- Use of segment registers make a program address insensitive. Here all memory addressing is performed using displacement (offset). Starting memory address is stored in the segment register and the actual address is given by :

Scanned with CamScanner

Contents of the segment register + address of the operand in the instruction.

The address of the variable X can be found as given below :

500 + 30 = 530

Segment register offset Actual address

- The BSS loader allows many code segments and one common data segment. The assembler assembles each code segment independently and passes on to the loader the following :
  1. Object program prefixed by a transfer vector that consists of addresses containing names of subroutines referenced by the source program.
  2. Relocation information, i.e. locations in the program that must be changed if it is loaded in an arbitrary location in the main memory.
  3. Length of the source program and the length of the transfer vector.
- The loader loads the transfer vector and the object code into memory, then the loader loads each subroutine identified in transfer vector. The transfer vector is used to solve the problem of linking and the program length information is used to solve the problem of allocation.

Syllabus Topic : Direct Linking Loaders

5.2.6 Direct Linking Loader

(May 2013)

Q. In case of a Direct Linking Loader, what is the information required to be passed by a translator to the loader. **SPPU - May 2013, 4 Marks**

It is a general relocatable loader, and is perhaps the most popular loading scheme presently used.

- It is a relocatable loader
- It allows multiple procedure segments and multiple data segments.

The assembler must give the loader the following information with each procedure or data segment :

1. The length of segment.
2. A list of symbols defined in the current segment that may be referenced by other segments – public declaration.
3. A list of all symbols not defined in the segment but referenced in the segment – external variables.
4. Information about address constants
5. The machine code translation of source program and the relative addresses assigned.

The object module produced by the assembler is divided into 4 sections :

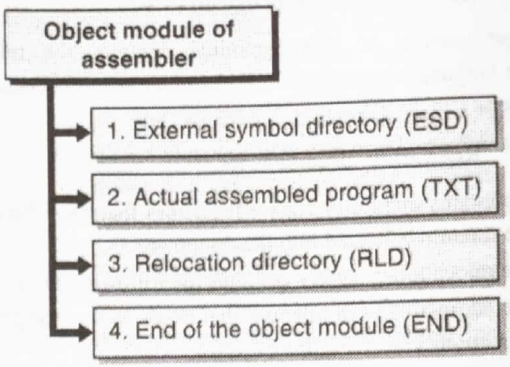


Fig. C5.2 : Object module of assembler

1. External symbol directory (ESD)

It contains information about all symbols that are defined in this program but may be referenced by other programs. It also contains symbols referenced in this program but defined in other program. A sample ESD is shown in Fig. 5.2.6.

				LC
1.	MAIN	START		0
2.		ENTRY RESULT		–
3.		EXTRN SUM		–
				–
				–
				–
				–
				–
				–
				–
				–
				–
				–
				–
10.	RESULT	DS	4	32
		END		36

Fig. 5.2.6(a) : A sample source program

Line No.	Symbol	Type	Relative location	Length
1.	MAIN	SD	0	36
2.	RESULT	LD	32	–
3.	SUM	ER	–	–

Fig. 5.2.6(b) : ESD for program shown in Fig. 5.2.6(a)

SD – symbol is a segment definition

LD – symbol is defined in this program but it can be referenced by other programs

ER – symbol is an external reference. It is defined in some external program.

- The relative location of the program (MAIN) is 0 and its size is 36
- The symbol RESULT is locally defined (LD) and its relative address is 32.
- The symbol SUM is an external reference (ER).

Scanned with CamScanner

2. Actual assembled program (TXT) :

Text portion of object module contains the relocatable machine language instructions and data that were produced during translation.

3. Relocation directory (RLD) :

It contains one entry for each address that must be changed when the module is loaded into main memory.

The relocation directory contains the following information :

1. The address of each operand that needs to be changed due to relocation
2. By what it has to be changed
3. The operation to be performed

4. End of object module (END) :

This indicated the end of object module.

Example 5.2.1

With respect to loader functions state whether the following statements are true or false.

(i) In absolute loader relocation is done by assembler.

(ii) In absolute loader linking is done by programmer.

(iii) In compiler-and-go loader allocation is done by assembler.

(iv) In compile-and-go loader, loading is done by loader.

Solution :

(i) True (ii) True (iii) True

(iv) False : Everything is handled by the assembler/compiler.

Example 5.2.2

Compare compile-and-go loader and absolute loader.

Solution :

Absolute loader is for non-relocatable program :

1. It accepts output of assembler as input.
2. It avoids re-assembly of program, every time it is executed.
3. Assembler need not reside in the memory at the time of execution. Loader is required to be in the memory but it is much smaller compared to an assembler.
4. The four loading functions are accomplished in an absolute loading scheme :
   1. Allocation – by programmer
   2. Linking – by programmer
   3. Relocation – by assembler
   4. Loading – by loader.

Compile-and-go loading scheme is useful if the program is to run a few times. In this scheme, everything is done by the assembler.

- A portion of the memory is wasted as it is occupied by the assembler.
- It is required to re-translate the user program every time it is run.
- It is difficult to handle multiple segments. Thus, it becomes very difficult to develop a modular program under this scheme.

Example 5.2.3

At what point of time do each of the following loading schemes perform binding.

(i) Direct linking loader

(ii) Absolute loader

Solution :

- In absolute loader linking is done by the programmer at the time of writing of the program.
- In direct linking loader, binding is done twice :
  1. At the time of linking.
  2. At the time of loading.

Example 5.2.4

State whether the following statements are true or false and justify your answer.

(i) Relocation is performed by linker.

(ii) Transfer vector is used by direct linking loader.

(iii) In absolute loader, linking is done by programmer.

(iv) In compile-and-go loader linking is performed by loader.

Solution :

1. True, translated addresses are changed by the linker at the time of linking.
2. True, transfer vector consists of addresses containing names of subroutines referenced by the source program. The statement corresponding to a subroutine call is translated into a transfer instruction indicating a branch to the location of the transfer vector associated with the subroutine.
3. True, absolute loader does not perform linking. It simply places the executable program at the pre-assigned addresses in the memory.
4. No, in compile-and-go scheme loading functions are performed by the assembler.

Example 5.2.5

State true or false :

(i) Loader loads and executes object code.

(ii) Linkers and loaders are not needed with re-locatable programs.

(iii) Transfer vector is used by direct linking loader.

(iv) In absolute loader relocation is done by assembler.

Solution :

(i) True (ii) False (iii) True (iv) True.

Example 5.2.6

Suppose that a programming language requires an interpreter, what kind of loader scheme is required in this case ? Justify your answer.

Solution : Normally, linking is not performed by an interpreter. It does the followings :

1. Gather function arguments and makes sure they are valid.
2. Calls the function
3. Handles the return value.

At the best, it can be said that interpreter uses compile-and-go scheme. Interpreter has to reside in the memory at the time of execution.

Scanned with CamScanner

Example 5.2.7

What is linkage editor ? What is the essential difference between linkage editor and linking loader ?

Solution :

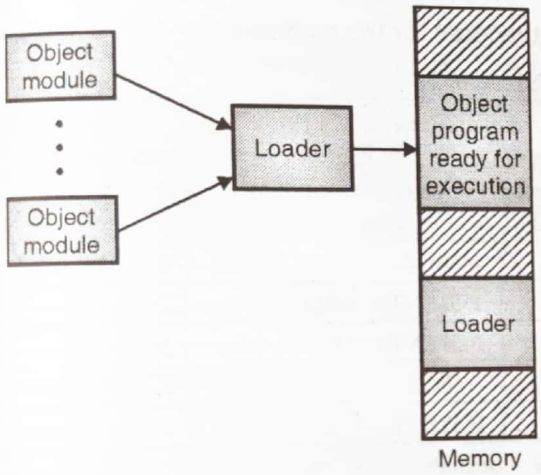
A linkage editor is a linker that links assembled programs. The linker resolves references between program modules and libraries of subroutines. The output of a linkage editor is load module, which is executable code ready to run.

This executable module can be loaded by a loader in the main memory after allocation and relocation.

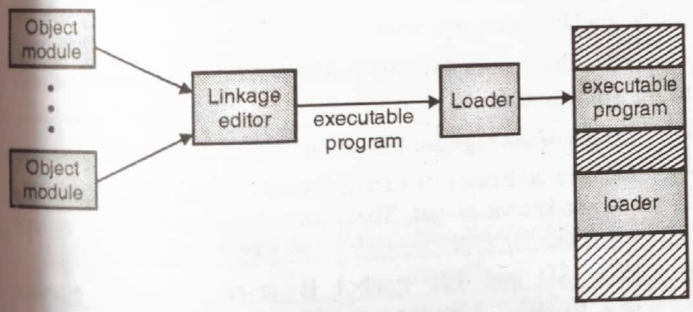
On the other hand, a linking loader performs both linking and loading.

The major disadvantage of linking loader is that every time you want to run the program, linking must be carried out. This step is performed only once in linkage editor.

The two schemes are shown in the Fig. Ex. 5.2.7.



(S4.22(a)) **Fig. Ex. 5.2.7(a) : Linking loader scheme**



(S4.22(b)) **Fig. Ex. 5.2.7(b) : Linkage editor scheme**

Syllabus Topic : Design of an Absolute Loader

5.3 Design of an Absolute Loader

→ **(Dec. 2013)**

Q. Explain absolute loaders. SPPU - Dec. 2013, 4 Marks

In absolute loading scheme the programmer and the assembler perform the following tasks :

1. Allocation 2. Relocation 3. Linking

The absolute loader reads the object program line by line and moves the text of the program into the memory at the location specified by the assembler.

The object program generated by the assembler must communicate the following information to the loader :

1. It must convey the machine instructions that the assembler has created along with the memory address.
2. It must convey the starting execution point. Program execution will start at this point after the program is loaded.

The object program is a sequence of object records. Each object record specifies some specific aspect of the program in the object module. There are two types of records :

1. Text record containing binary image of the assembly program.
2. Transfer record containing the starting (entry) point of execution.

The formats of text and transfer records are given in Fig. 5.3.1

Record Type	Number of bytes of information	Memory address	Binary image of data or instruction

(a) Text record

Record Type	Number of bytes of information = 0	Address of the entry point

(b) Transfer record

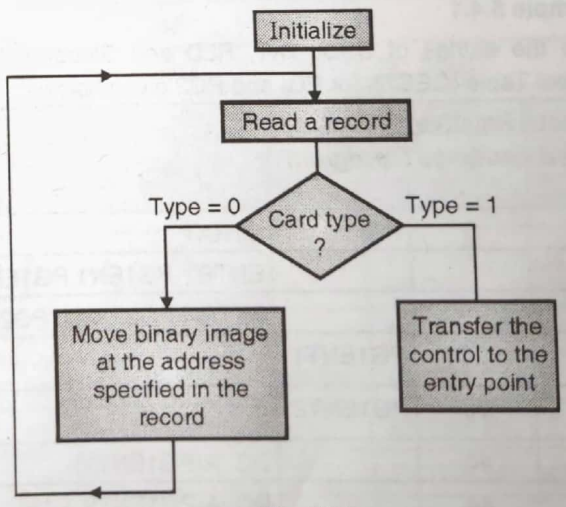
Record type = 0 for Text record

Record type = 1 for Transfer record

Fig. 5.3.1 : Formats of object records

☞ **Algorithm :** The algorithm for absolute loader is very simple. The loader reads the object file record by record and moves the binary image at the locations specified in the record. The last record is the transfer record. On reaching the transfer record the control is transferred to the entry point for execution.

☞ **Flowchart :** Flowchart is given in Fig. 5.3.2.



(S5.8) **Fig. 5.3.2**

Syllabus Topic : Design of Linker

5.4 Design of Direct Linking Loader

→ **(May 2015)**

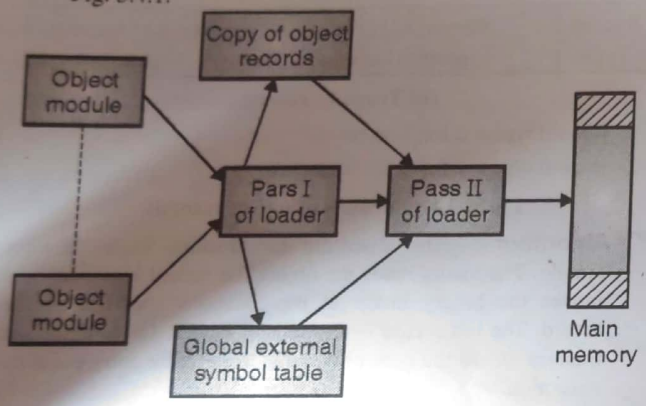
Q. Explain design of direct linking loader. Also explain the required data structures. SPPU - May 2015, 7 Marks

Scanned with CamScanner

The design of a direct linking loader is more complicated than that of the absolute loader. The input to the loader is set of object programs (generated by assembler/compiler) to be linked together. Each object module is divided into 4 sections :

1. External symbol directory (ESD).
2. Actual assembled program containing the binary code (TXT)
3. Relocation directory (RLD).
4. End of the object module (END).

- A direct linking loader requires two passes to complete the linking process.
  - (i) Pass I, assigns addresses to all external symbols.
  - (ii) Pass II, performs actual loading, relocation and linking.
- In pass I, a global external symbol table (GEST) is prepared. It contains every external symbol and the corresponding absolute address value.
- The two-pars direct linking loader scheme in shown in Fig. 5.4.1.



(S5.9) **Fig. 5.4.1 : Two pass direct linking loader scheme**

Example 5.4.1

Write the entries of ESD, TXT, RLD and Global External Symbol Table (GEST), for PGI and PG2 given below :

Object record No.	Relative address	Source program	
1	0	PGI	STRAT
2			ENTRY PG1EN1,PG1ENT2
3			EXTRN PG2ENT1, PG2
4	20	PG1ENT1	=
5	30	PG1ENT2	=
6	40		DC A(PG1ENT1)
7	44		DC A(PG1ENT2 + 15)
8	48		DC A(PG1ENT2 – PG1ENT1 – 3)
9	52		DC A(PG2)
10	56		DC A(PG2ENT1 + PG2 – PG1ENT1 + 4)
11	60		END
12	0	PG2	START
13			ENTRY PG2ENT1
14			EXTRN PG1ENT1, PG1ENT2
15	16	PG2ENT1	=
16	24		DC A(PG1ENT1)
17	28		DC A(PG1ENT2 + 15)
18	32		DC A(PG1ENT2 – PG1ENT1 – 3)
19	36		END

Solution :

Object records for the module PG1

1. ESD records

Source object record No.	Name	Type	ID	Relative address	Length
1	PG1	SD	01	0	60
2	PG1ENT1	LD	–	20	–
2	PG1ENT2	LD	–	30	–
3	PG2	ER	02	–	–
3	PG2ENT1	ER	03	–	–

SD – Segment definition

LD – Local definition, declared in the current segment but can be used by other segments.

ER – External definition, declared in other segments but can be used in this segment.

- Length of the segment PG1 is 60
- Relative addresses of external symbols PG2 and PG2ENT1 are not known as yet. They are not declared in the current segment.
- Each SD and ER symbol is given a unique number (e.g. 01, 02, ...) by the assembler. This ID will be used in RLD.

2. TEXT records :

We will confine ourselves in showing entries involving address constants.

Source object record No.	Relative address	Contents	Comments
6	40 – 43	20	Address of PG1ENT1 is 20
7	44 – 47	45	Address of PG1ENT2 + 15 = 30 +15 = 45
8	48 – 51	7	PG1ENT2 – PG1ENT1 – 3 = 30 – 20 – 3 = 7
9	52 – 55	0	Address of PG2 is not known. It is taken as 0.

Scanned with CamScanner

Source object record No.	Relative address	Contents	Comments
10	56 – 59	– 16	PG2ENT1 + PG2 – PG1ENT1 + 4 = 0 + 0 – 20 + 4 = – 16 Addresses of PG2ENT1 and PG2 are not known

3. RLD records

Relocation of variables can be handled through segment registers. They do not appear in RLD records. RLD records are for address constants.

– Difference of two relative addresses is a constant. Therefore, the entry

8 48 DC A (PG1ENT2 – PG1ENT1 – 3)

will not appear in RLD records.

Source object record No.	ESD ID	Length (in bytes)	Flag + or –	Relative address
6	01	4	+	40
7	01	4	+	44
9	02	4	+	52
10	03	4	+	56
10	02	4	+	56
10	01	4	–	56

Object records for module PG2

1. ESD records

Source object record No.	Name	Type	ID	Relative address	length
12	PG2	SD	01	0	36
13	PG2ENT1	LD	–	16	–
14	PG1ENT1	ER	02	–	–
14	PG1ENT2	ER	03	–	–

2. TEXT records

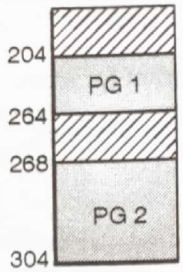
Source object record No.	Relative address	Contents
16	24 – 27	0
17	28 – 31	15
18	32 – 35	– 3

3. RLD records

Source object record No.	ESD ID	Length (in bytes)	Flag + or –	Relative address
16	02	4	+	24
17	03	4	+	28
18	03	4	+	32
18	02	4	–	32

Global External Symbol Table (GEST)

The GEST contains every external symbol and its corresponding assigned memory address. If we assume that PG1 is loaded at location 204 and PG2 is loaded at location 268. It is shown in Fig. Ex. 5.4.1.



(S5.10)**Fig. Ex. 5.4.1**

External symbol	Assigned memory address
PG1	204
PG1ENT1	224
PG1ENT2	234
PG2	268
PG2ENT1	284

Example 5.4.2

Show ESD, RLD and TXT cords for the source code given below. Also show the contents of GEST table LESA table. Finally show the load map.

Source cord reference	Relative address		Source program
1	0	MAIN	START
2			ENTRY M1, M2
3			EXTRN PROG, P1, P2
4	30	M1	–
5	40	M2	–
6	50		DC A(M2)
7	54		DC A(M1 + 5)
8	58		DC A(M2 – M1 – 4)
9	62		DC A(PROG)
10	66		DC A(P1 + P2 – PROG)
11			END
12	0	PROG	START
13			ENTRY P1, P2
14			EXTRN M1, M2
15	16	P1	–
16	26	P2	–
17	36		DC A(M1)
18	40		DC A(M2 + 10)
19	44		DC A(P2 – P1 – 3)
20			END

Assume initial program load address as 400.

Scanned with CamScanner

Solution :

1. Object deck program for main.

ESD Card					
Source card reference	Name	Type	ID	Relative address	Length
1	MAIN	SD	01	0	70
2	M1	LD	–	30	–
2	M2	LD	–	40	–
3	PROG	ER	02	–	–
3	P1	ER	03	–	–
3	P2	ER	04	–	–

TXT Card			
Source card reference	Relative address	Contents	Comments
6	50–53	40	Address of M2 = 40
7	54–57	35	M1 + 5 = 30 + 5
8	58–61	6	M2 – M1 – 4 = 40 – 30 – 4
9	62–65	0	External reference
10	66–69	0	External reference

RLD card				
Source card reference	ESD ID	Length (in bytes)	Flag + or –	Relative address
6	01	4	+	50
7	01	4	+	54
9	02	4	+	62
10	03	4	+	66
10	04	4	+	66
10	02	4	–	66

2. Object deck program for PROG.

ESD card					
Source card reference	Name	Type	ID	Relative address	Length
12	PROG	SD	01	0	48
13	P1	LD	–	16	–
13	P2	LD	–	26	–
14	M1	ER	02	–	–
14	M2	ER	03	–	–

TXT card			
Source card reference	Relative address	contents	Comment
17	36 – 39	0	External reference
18	40 – 43	10	M2 + 10 = 0 + 10 = 10
19	44 – 47	7	P2 – P1 – 3 = 26 – 16 – 3 = 7

RLD card				
Source card reference	ESD ID	Length (in bytes)	Flag + or –	Relative address
17	01	4	+	40
18	01	4	+	44

3. Global External Symbol Table (GEST) :

Since the initial load address is 400, MAIN will be loaded from the initial address. The segment PROG will be loaded from Initial load address + size of the segment,

MAIN = 400 + 70 = 470

GEST Table	
External symbol	Assigned memory address
MAIN	400
M1	430
M2	440
PROG	470
P1	486
P2	496

4. Local External Symbol Array (LESA)

LESA is created for each segment. It contains SD and LD type entries of each segment.

	LESA of MAIN
	Assigned memory addresses
1	400
2	430
3	440
⋮	

	LESA of PROG
	Assigned memory addresses
1	470
2	486
3	496

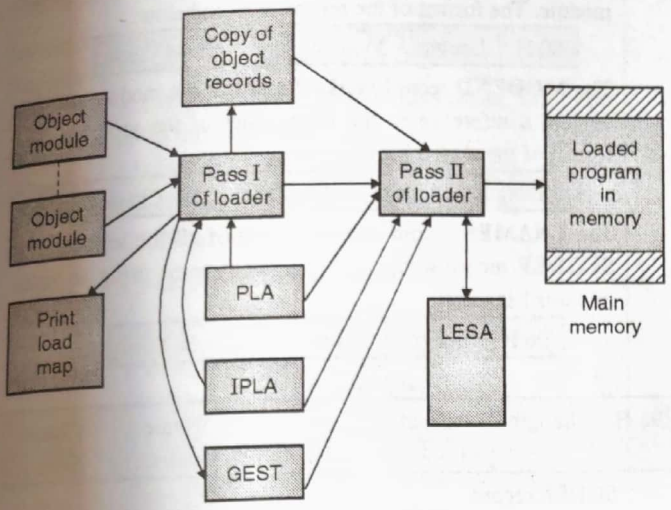
5.4.1 Format of Databases of a Direct Linking Loader

Major databases in the design of a direct linking loader are :

1. Object records : Object records are of four types. These are
   (a) ESD (b) TXT
   (c) RLD (d) END.
   It is input to the loader.
2. Global External Symbol Table (GEST)
3. Local External Symbol Table (LESA)
4. Program Load Address (PLA)
5. Initial Program Load Address (IPLA)

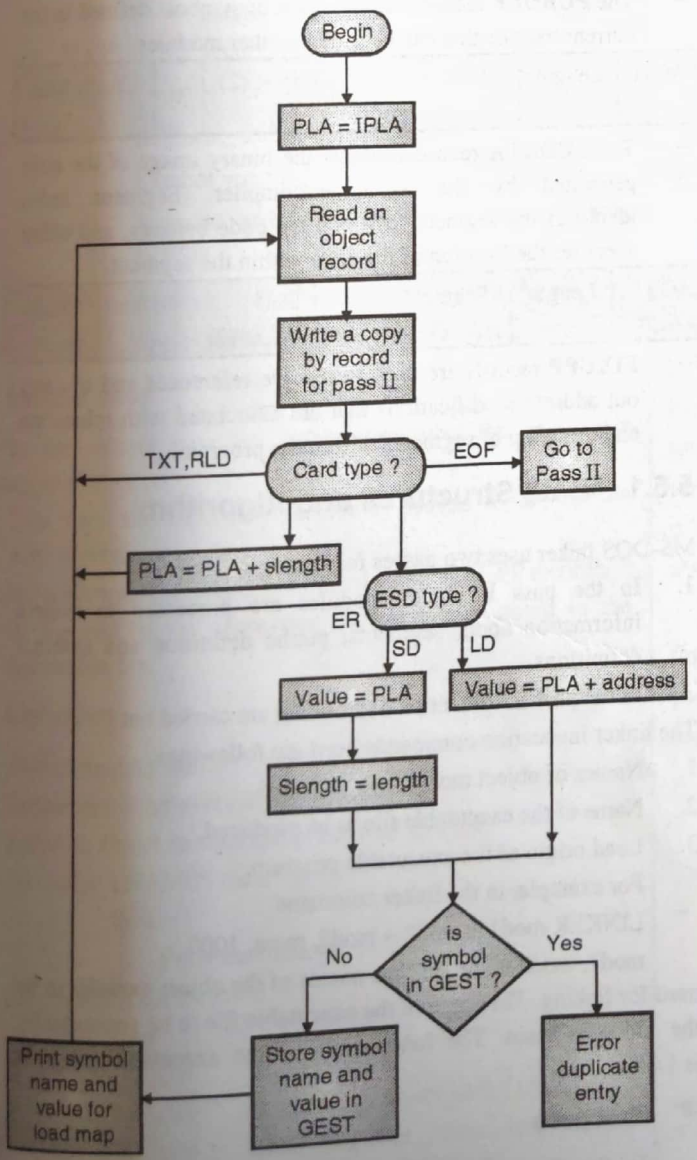
Use of databases by the loader is shown in Fig. 5.4.2.

Scanned with CamScanner



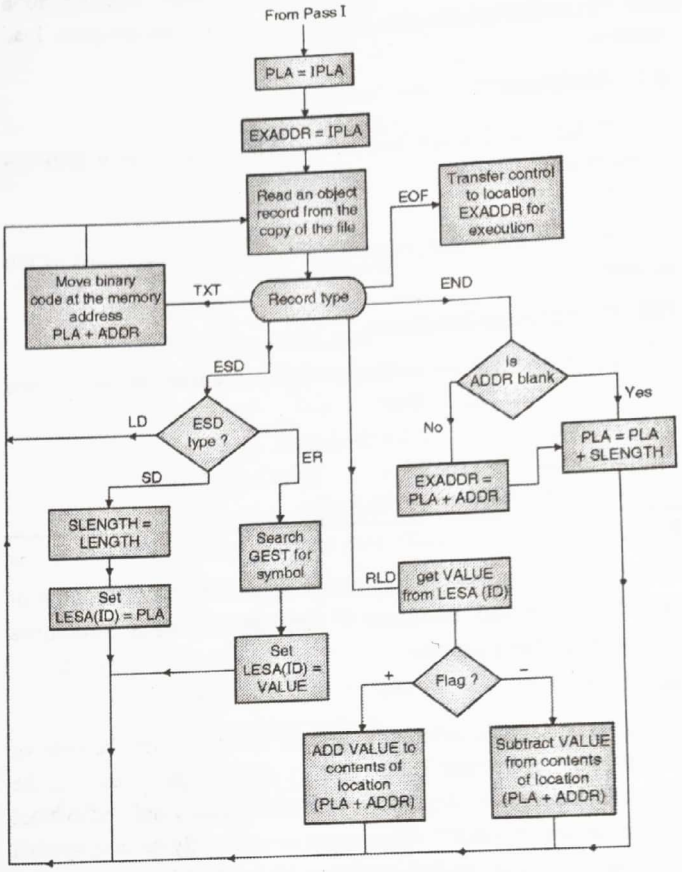
(S5.11)**Fig. 5.4.2 : Use of data bases by loader**

5.4.2 Flowchart for Pass I



(S5.12)**Fig. 5.4.3**

5.4.3 Flowchart for Pass II



(S5.13)**Fig. 5.4.4**

Example 5.4.3

Answer the following with respect to two pass linking loader :

(i) What is the function of Pass-I ?

(ii) What is the function of Pass-II ?

(iii) Suppose you were to restrict to one pass loader, what facilities would you be able to give to the user (viz : Simple address relocation, external symbol etc.)

Solution :

(i) Pass I assigns addresses to external symbols.

(ii) Pass II performs actual loading, allocation, relocation and linking.

(iii) One pass loader will not be able to handle external symbols. Other activities like allocation, relocation, loading and linking can be handled by one pass loader.

Example 5.4.4

Give an example of the following types of address constants:

(i) Simple re-locatable (ii) Absolute

(iii) Complex re-locatable.

Solution :

(i) Simple re-locatable

If **L1** is a relocatable symbol in a program segment and C1 is a constant then

$L1 \pm C1$

Scanned with CamScanner

the address is constant, A (L1 ± C1) is simple relocatable. This address can be calculated by the assembler. The relocation will be performed by the loader by adding the program load address.

(ii) Absolute

If **L1** and **L2** are two relocatable symbols in a program segment then

L1 – L2

will be an absolute value. This value can be calculated by the assembler.

(iii) Complex re-locatable

If **L1** and **L2** are defined in other program segments then assembler cannot handle them.

In such cases

L1 ± L2

must be calculated by the loader.

Example 5.4.5

How can a linker resolve symbols defined to be synonyms of externally defined symbols ? Explain the data structures required for this purpose.

Solution :

Linker handles externally defined symbols with the help of Global External Symbol Table (GEST). GEST is prepared from the object deck program. Any synonyms of externally defined symbol will not appear in GEST. A synonym of externally defined symbol is treated as a local symbol and it can be handled by the assembler with the help of symbol table.

5.5 Implementation of MS DOS Linker

→ **(May 2014)**

Q. Explain MS-DOS linker in detail.

SPPU - May 2014, 4 Marks

MS DOS compilers and assembler produce object modules. The object module is a sequence of object records. There are 14 types of object records. These records contain the following three basic categories of information :

1. Binary image (machine instructions)
2. External references
3. Public definitions.

The names and purpose of the object record types is given below :

Record Type	Id (Hey)	Description
THEADER	80	Translator header
LNAMES	96	List of names record
SEGDEF	98	Segment definition
EXTDEF	8C	External definition
PUBDEF	90	Public definition
LEDATA	A0	Translated instructions and data
FIXUPP	9C	Fixup records for relocation
MODEND	8A	Module end record.

- The **THEADER** record specifies the name of the object module. The format of the record is given below.

80 H	Length	Module name	Check-Sum

- The **MODEND** record marks the end of the module and can contain a reference to the entry point of the program. The format of the record is given below.

80 H	Length	Type (1)	Start addr. (5)	Check-Sum

- The **LNAMES** record contains a list of all the segments. A **SEGDEF** record designates a segment name using an index into this list.

96 H	Length	Name list	Check-Sum

LNAMES record

98 H	Length	Attributes (1 – 4)	Segment length (2)	Name index (1)	Check-Sum

SGDEF record

- The **EXTDEF** record contains a list of external symbols that are used in this module.

8 CH	Length	External reference list	Check-Sum

- The **PUBDEF** record contains a list of symbols defined in the current module that can be used by other modules.

90 H	Length	Base (2 – 4)	Name	Offset (2)	...	Check-Sum

- The **LEDATA** record contains the binary image of the code generated by the assembler/compiler. Segment index identifies the segment to which the code belongs, and offset specifies the location of the code within the segment.

A0 H	Length	Segment index (1 – 2)	Data offset (2)	Data	Check-Sum

- **FIXUPP** records are used to resolve references and to carry out address modifications that are associated with relocation and grouping of segments within the program.

5.5.1 Data Structures and Algorithm

MS-DOS linker uses two passes for linking :

1. In the pass I, object modules are processed to collect information about segments, public definition and external definitions.
2. In the pass II, relocation and linking are carried out.

The linker invocation commands need the followings :

1. Names of object modules to be linked
2. Name of the executable file to be produced.
3. Load origin of the executable program.

For example, in the linker command

LINKER mod1 + mod2 + mod3, main, 1000

mod1, mod2 and mod3 are names of the object modules to be used for linking. The name of the executable file to be generated by the linker is main. The load origin of the executable program is 1,000.

☞ First pass

In the first pass, linker processes the object records for building external symbol director (ESD).

Scanned with CamScanner

Algorithm

1. Extract load_origin from the command line.
2. Repeat step 3 for each module to be linked.
3. Select the next object module from the command line. For each record in the object module :
   (a) If an LNAMES record then enter the name of the module in the name directory (NAMED).
   (b) If a SEGDEF record then
      (i) i = name index from the record
      (ii) segment_name = NAMED [i]
      (iii) If an absdute segment then enter (segment_name, segment_addr) in ESD.
      (iv) If the segment is relocatable then
         - Align load_origin with the next paragraph. It should be multiple of lb.
         - Enter (segment_name, load_origin) in ESD.
         - Load_origin = load_origint segment length.
   (c) If a PUBDEF record then
      (i) i = base
      (ii) Segment_name = NAMED [i]
      Symbol = name
      (iii) Segment_addr = load address of segment_name in ESD
      (iv) Symbol_addr = segment_addr + offset
      (v) Enter (symbol, symbol_addr) in ESD.

Second pass

In the second pass the linker constructs the executable program in work-area.

- Data from the LEDATA records is moved to appropriate parts of work area.
- Relocation and linking is performed using FIXUPP records.
- At the end of the processing, the output is stored in the executable file.

Algorithm

1. Extract module names from the command.
2. Repeat step 3 for every object module.
3. Select an object module and process its records.
   (a) If an LNAMES record
      then
         Enter the names in NAMELIST.
   (b) If a SEGDEF record
      then
         (i) i = name index
         (ii) segment _name=NAMELIST[i]
         (iii) Enter (segment _name, load address from ESD) in SEGTAB
   (c) If an EXTDEF record
      then
         (i) external_name=name from EXTDEF record
         (ii) Enter (external_name, load address from ESD) in EXTTAB.
   (d) If an LEDATA record then
      (i) i = segment index
      (ii) d = data offset
      (iii) program_load_origin=SEGTAB[i].load address
      (iv) address_in_work_area = address of work_area + program_load_orign - <load orign> + d
      (v) Move data from LEDATA into the memory area starting at the address address_in_work_area.
   (e) If a FIXUPP record, then for each FIXUPP specification
      (i) f = offset from locat field
      (ii) Perform required fix up.
   (f) If a MODEND record then
      If the start address is specified, compute the corresponding load address.

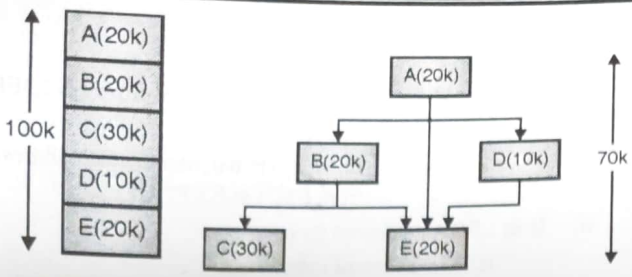
5.6 Overlay Structure

An overlay is a part of a program which has the same load origin as some other parts of the program. It is normally used in a system which does not support virtual memory. In static linking/loading, it is necessary that all the subroutines needed for execution are loaded into the main memory at the same time. If the total amount of memory required by all these subroutines exceeds the amount available, as is common with large programs on small computers, we can get rid of this problem with the help of overlays.

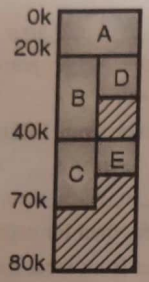
Usually the subroutines of a program are needed at different times. By finding which subroutines call other subroutines it is possible to produce an overlay structure that identifies mutually exclusive subroutines.

- The Fig. 5.6.1 illustrates a program consisting of five subprograms (A, B, C, D and E) that require 100k bytes of memory.
- The arrow indicates that subprogram A only calls B, D and E. The subprogram B only calls C and E. The subprogram D only calls E.
- Procedures B and D are never in use at the same time. Similarly, C and E are never in use at the same time.
- We can create an overlay structure as shown in the Fig. 5.6.1. It will require 70 k of memory.

Scanned with CamScanner



(a) without overlays **(b) Dynamic loading**



(c) Possible storage allocation with overlays

(S4.23)**Fig. 5.6.1 : Storage allocation with overlays**

- In order for the overlay structure to work it is necessary for the module loader to dynamically load the various subroutines as they are needed.
- The portion of the loader that actually intercepts the calls and loads the necessary subroutines is called overlay manager.
- This scheme is called dynamic loading.

5.7 Software Tools for Program Development

Software tools are used extensively for program development and user interfaces. Here, we will discuss same important tools like:

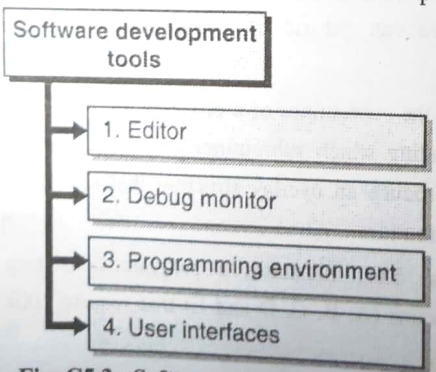


Fig. C5.3 : Software development tools

➜ 5.7.1 Editor ➜ (Oct. 2016)

Q. Explain in details design of editor.

SPPU - Oct. 2016(In Sem), 6 Marks

Editor is a computer program that allows a user to create and revise a target document. The document includes objects such as computer programs, text, equations, tables, diagrams, lines and photographs. A text editor is a program in which the primary elements being edited are character strings of the target strings. The document editing process is an interactive user_computer dialogue designed to accomplish four tasks :

1. Select the part of the document to be viewed and manipulated.
2. Determine two to format this view on line and how to display it.
3. Specify and execute operations that modify the target document.
4. Update the view appropriately.

Text editors come in the following forms :

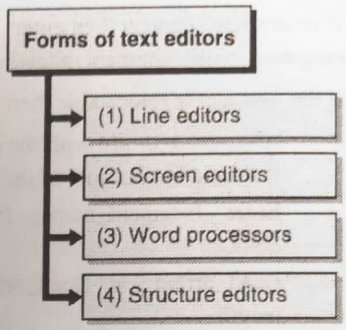


Fig. C5.4 : Forms of text editors

➜ (1) Line editors

It is a primitive form of editor. It requires you to specify a specific line of text before you can make changes to it. A line editor does not display the text in the manner it would appear if printed.

- In a line editor, the cursor cannot be used to move around and select a portion of the document.
- Any modification is restricted to current line.
- Interaction with the editor is through a set of commands. This makes a line editor a less user friendly.

➜ (2) Screen editors

Screen editor enables one to modify any text that appears on the display screen by moving the cursor to the desired location. The editor displays a screen full of text at a time. The user can see the effect of edit operation on the screen. This is very useful while formatting the text to produce printed documents.

➜ (3) Word processors

A word processor recognises word as a basic entity. It is possible to support a spell-check option. A word processor has the concept of paragraph. A typical word processor has the following options :

1. Moving a section of text from one place to another.
2. Deleting a section of text.
3. Searching a word.
4. Merging of text
5. Replacement of word.
6. Spell-check.

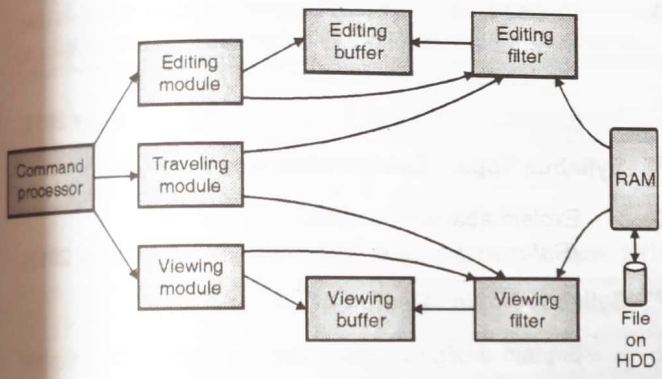
➜ (4) Structure editors

A structure editor has awareness of the structure of document. The structure is specified by the user while creating or modifying the document. Syntax directed editor is an example of structure editor.

Scanned with CamScanner

5.7.1(a) Editor Structure

- Many text editors have a structure similar to the Fig. 5.7.1.
- The common language processor accepts input from the user's input devices, and analyses it and its syntactic structure.
- Editing module is a collection of modules dealing with editing tasks. The current editing pointer is maintained by the editing module.
- The travelling module of the editor performs the setting of the current editing and viewing pointers.
- In viewing document, the start of the area to be viewed is determined by the current viewing pointer. This pointer is maintained by the viewing module.



(S6.1)**Fig. 5.7.1 : Typical editor structure**

- When the user issues an editing command, the editing module invokes the editing filter. This component filters the document to generate a new editing buffer based on the current editing pointer as well as on the editing filter parameters.

➜ 5.7.2 Debug Monitor

An interactive debugging system provides programmers with facilities that aid in the testing and debugging of programs. Debug monitors provide the following facilities for dynamic debugging :

1. Setting breakpoints in the program.
2. Initiating a debug conversation when control reaches the break point.
3. Displaying values of variables.
4. Assigning new values to variables.

A debugging system should provide functions such as tracing and traceback. Tracing can be used to track the flow of execution logic and data modification. Traceback can show the path by which the current statement is reached.

It is also important for a debugging system to have good program display capabilities. It must be possible to display the source program being debugged, complete with statement numbers.

While debugging a program, user specifies a list of breakpoints and actions to be performed at breakpoints. The debug monitor builds a table containing statement number and debug action.

➜ 5.7.3 Programming Environment

Programming environment is an integrated program development environment where most of the program development facilities are available number the same environment. These facilities include :

1. Program creation.
2. Editing.
3. Execution.
4. Testing and debugging.

A programming environment consists of the following components

1. A syntax directed editor.
2. Compiler, interpreter.
3. Debug monitor
4. Dialog monitor

These components are accessed through dialog monitor. The syntax directed editor understands the syntax of the source language. The editor can do syntax analysis and convert the input statement into an intermediate representation. The program execution or interpretation can be carried out, immediately after the program is keyed in. In case of any error, the programmer can enter the debug mode to fix the bug. The main advantage is easy accessibility of all functions through dialog monitor.

➜ 5.7.4 User Interfaces

A user interface (UI) plays an important role in simplifying the interaction of a user with an application. UI functioning has two important aspects :

1. Issuing of commands
2. Exchange of data

A UI can be seen as consisting of two components :

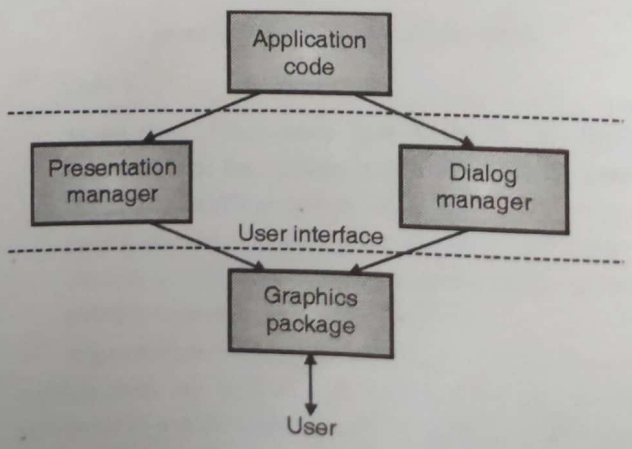
1. A dialog manager.
2. A presentation manager

A typical UI schematic is shown in Fig. 5.7.2.

The presentation manager is responsible for managing user's screen and for accepting data and presenting results.

The dialog manager is responsible for interpreting user command and implementing them by invoking different modules of the application code.

The dialog manager is also responsible for error messages and on line help functions and for organizing changes in the visual context of the user.



(S6.2)**Fig. 5.7.2 : User interface**

Scanned with CamScanner

5.8 Exam Pack (University Questions)

Q. What is loader ? List basic functions and features of a loader. *(Refer section 5.1)* (8 Marks)

(May 2014, Dec. 2015)

Q. Explain the terms : Loader, Linker. *(Refer section 5.1)* (4 Marks) **(Dec. 2014)**

Q. Explain functions of a Loader. *(Refer section 5.1)* (4 Marks) **(Aug. 2015(In Sem))**

Q. What is linker ? *(Refer section 5.1)* (4 Marks)

(May 2014)

Q. Why program relocation is required and how is it performed? *(Refer section 5.1)* (6 Marks)

(May 2013)

Example 5.1.1 **(Dec. 2014, Oct. 2016(In Sem), May 2017)**

☞ **Syllabus Topic : Loader Schemes**

Q. List and explain the different loader scheme in brief. *(Refer section 5.2)* (8 Marks)

(May 2014, Dec. 2014, Dec. 2015, Dec. 2016)

☞ **Syllabus Topic : Compile and go**

Q. Explain any one type of loader. *(Refer section 5.2.1)* (3 Marks) **(Dec. 2014)**

Q. Explain compile and go-loader scheme. *(Refer section 5.2.1)* (6 Marks)

(Dec. 2015, Oct. 2016(In Sem))

☞ **Syllabus Topic : Absolute Loaders**

Q. Explain Absolute Loaders loading scheme. *(Refer section 5.2.3)* (6 Marks)

(Aug. 2015(In Sem))

☞ **Syllabus Topic : Subroutine Linkages**

Q. Explain sub routine linkers. *(Refer section 5.2.4)* (4 Marks) **(Dec. 2013)**

☞ **Syllabus Topic : Relocating Loaders**

Q. Explain relocation loaders. *(Refer section 5.2.5)* (4 Marks) **(Dec. 2013)**

☞ **Syllabus Topic : Direct Linking Loaders**

Q. In case of a Direct Linking Loader, what is the information required to be passed by a translator to the loader. *(Refer section 5.2.6)* (4 Marks)

(May 2013)

☞ **Syllabus Topic : Design of an Absolute Loader**

Q. Explain absolute loaders. *(Refer section 5.3)* (4 Marks) **(Dec. 2013)**

☞ **Syllabus Topic : Design of Linker**

Q. Explain design of direct linking loader. Also explain the required data structures. *(Refer section 5.4)* (7 Marks) **(May 2015)**

Q. Explain MS-DOS linker in detail. *(Refer section 5.5)* (4 Marks) **(May 2014)**

Q. Explain in details design of editor. *(Refer section 5.7.1)* (6 Marks) **(Oct. 2016(In Sem))**

Scanned with CamScanner

UNIT II

CHAPTER 6

Linkers

Syllabus Topics

Relocation and linking concepts, Design of linker, Self relocating programs, Static and dynamic linker.

Syllabus Topic : Relocation and Linking Concepts

6.1 Relocation and Linking Concept

6.1.1 Linking

➔ **(May 2013, May 2014, Dec. 2014)**

Q. Explain the need of a linker in program development. **SPPU - May 2013, 4 Marks**

Q. What is linker ? **SPPU - May 2014, 2 Marks**

Q. Explain the term linker. **SPPU - Dec. 2014, 4 Marks**

Any usable program written in any language has to use functions/ subroutines. These functions could be either user defined functions or they can be library functions.

For example, consider a program written in C-language. Such a program may contain calls to functions like printf() and scanf(). During program execution the main program as well as functions must reside in the main memory. In addition, every time a function is called, the control should get transferred to the appropriate function.

- The linking process makes address of modules known to each other so that transfer of control takes place during execution.
- Passing of parameters is handled by the linker. Parameters can be passed by value or by reference. A value returned by a function must be handled.
- Every public variable should have the same address in every module. An external variable can be defined in one module and can be used in another module. The address of an external variable should be same in every module. Resolving of addresses of symbolic references are handled by the linker.

6.1.2 Relocation

➔ **(May 2013)**

Q. Why program relocation is required and how is it performed? **SPPU - May 2013, 6 Marks**

Relocation is the process of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from any designated area of memory.

The statement

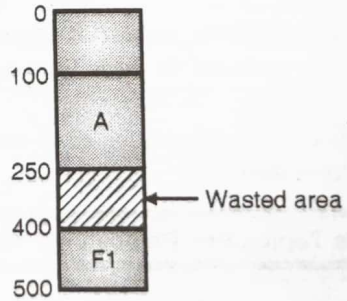
MOVER AREG, X

This is an address sensitive instruction. For this instruction to execute correctly, the actual address of X should be put in the instruction.

Assume that a program written in C-language (let us call it A) calls a function F1. The program A and the function F1 must be linked with each other. But when in main storage shall we load A and F1 ? A possible solution would be to load them according to the addresses assigned when they were translated.

☞ **Case I**

At the time of translation, A has been given storage area from 100 to 250 while F1 occupies area between 400 to 500. If we were to load these programs at their translated addresses, a lot of storage lying between them will be wasted.



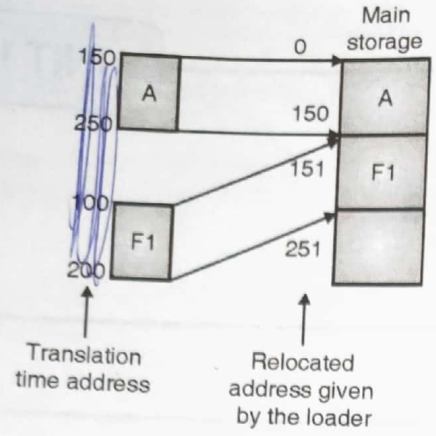
(S5.2)**Fig. 6.1.1 : Case I of relocation**

☞ **Case II**

At the time of translation, both A and F1 may have been translated with the identical start address 100. A goes from 100 to 250 and F1 goes from 100 to 200. These two modules cannot co-exist at same storage locations. The loader must relocate A and F1 to avoid address conflict or storage waste A possible relocation is shown in Fig. 6.1.2.

It may be noted that relocation is more than simply moving a program from one area to another in the storage. It refers to adjustment of address fields and not to movement of a program.

Scanned with CamScanner



(S5.3) **Fig. 6.1.2 : Relocation to avoid address conflict or storage waste**

6.1.3 Object Module

An assembler produces an object module. The object module is a sequence of object records. It contains all information necessary to relocate and link different object modules. It has four components :

1. Header
2. Binary image (machine instructions)
3. Relocation table
4. Linking table containing external references and public definitions.

- The header contains the followings :
  1. Size of the object module
  2. Name of the object module
  3. Translated origin
  4. Execution start address of the object module.
- The binary image contains the machine language program of the module.
- Relocation table contains the list of instructions requiring relocation. The operands in such instruction are address sensitive.
- Linking table contains information about external references and public declarations.

Syllabus Topic : Self Relocating Programs

6.2 Self Relocating Programs

→ **(Aug. 2015)**

Q. Explain the need of program relocatability.

SPPU - Aug. 2015(In Sem), 4 Marks

Program relocatability refers to the ability to load and execute a given program into an arbitrary place in memory as opposed to a fixed set of locations specified at program translation time. Depending on how and when the mapping from virtual address space to the physical address space takes place in given relocation scheme, there are two basic types of relocation :

1. Static 2. Dynamic

A self relocating program is a program which can perform the relocation itself. It contains the followings :

1. A table of information about address sensitive instructions in the program
2. Relocating logic that can perform the relocation of the address sensitive instructions.

Self relocating program contain the relocating logic that can perform the relocation of the address sensitive instructions. So there is no need of a linker in self-relocating programs.

Syllabus Topic : Static and Dynamic Linker

6.3 Static and Dynamic Link Libraries

6.3.1 Static Linking

A static linker takes object files produced by the compiler including library functions and produces an executable file. The executable file contains a copy of every subroutine (user defined or library function).

- The biggest disadvantage of the static linking is that each executable file contains its own copy of the library routines. If many programs containing same library routines are executed then memory is wasted.
- Another disadvantage of static linking is that newer versions of the library routines must be re-linked into executable.

6.3.2 Dynamic Linking

→ **(Oct. 2016)**

Q. Explain dynamic loading and linking.

SPPU- Oct. 2016(In Sem), 4 Marks

Dynamic linking defers much of the linking process until a program starts running. Dynamic linking involves the following steps :

1. A reference to an external module during run time causes the loader to find the target module and load it.
2. Perform relocation during run time.

There are several advantages to this approach :

1. Dynamically linked shared libraries are easier to create.
2. Dynamically linked shared libraries are easier to update.
3. Dynamic linking provides automatic sharing of code. If multiple applications require the same routine, operating system loads a single copy of the routine and links it with multiple applications.
4. It is easy to add new functionality to existing library. These functions could be useful in a variety of applications.
5. Dynamic linking permits a program to load and unload routines at runtime.

6.3.3 Dynamic Link Library (DLL)

DLL is Microsoft implementation of shared library in windows. The file formats for DLL are the same as windows EXE files. A DLL can contain :

1. Code 2. Data 3. Resources.

Scanned with CamScanner

With dynamic linking, shared code is placed into a single, separate file. The programs that call this file are connected to it at run time, with the operating system performing the linking.

While the DLL code may be shared, the data is private except where shared data is explicitly requested by the library. Each process using the DLL has its own copy of all the DLL's data. Sharing of DLL's data section allows interprocess communication through this shared memory.

Import libraries

Linking to dynamic libraries is usually handled by linking to an import library.

- The created executable file contains an import address in import address table (IAT). DLL function calls are referenced through this address. Each referenced DLL function contains its own entry in the IAT. At run-time, the IAT is filled with appropriate addresses that point directly to a function in the separately loaded DLL.
- The import libraries for DLL have · lib extension. For example, kernel32.dll, the primary dynamic library for window's base functions such as file creation and memory management, is linked via kernal32.lib.

Explicit run-time linking

It is possible to link an executable program with DLL by resolving addresses of the imported functions at the compile time.

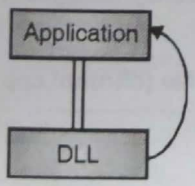
DLL files may be explicitly loaded and linked at run-time. Windows provides a set of standard API functions for it :

1. LoadLibrary : It loads a DLL file
2. GetProcAddress : It looks up the exported symbol by name and returns its address in IAT.
3. FreeLibrary : It unloads the DLL.

6.3.4 Call Back Functions

A callback function is a function which we write, but it is called by some other program or module, such as windows or DLL's.

For example, a DLL may control several clients, when a certain event occurs from the DLL, the callback function in the client is called.



(S6.14) **Fig. 6.3.1 : A DLL calling a function in an application via a call back**

- A callback is a function in an application that a DLL can call at suitable times. It is possible for a DLL to call a function in an application, provided that it has a pointer to such a function (Fig. 6.3.1).

Implementing the callback function

Implementation of callback function involves writing a DLL that will make use of the callback. This can be broken into five parts :

1. Defining the callback function
2. Declaring a type for the callback function
3. Writing the code that uses the callback
4. Implementing the function in the client
5. Call to DLL

The code in the **calling application** is simple. First, we must declare a function to be used as the callback :

```
#include "myDll.h"
Void CALLBACK MyCallback (int x)
   {
     //code of the function
   }
```

The address of the callback function must be passed to the DLL function. Let us assume that the DLL function is CalcResult.

The code to call CalcResult is simple :

CalcResult (MyCallback) ;

In the above code, we have passed a pointer to callback function as parameter.

Callback in the DLL

In the DLL, we must declare a function type and a function itself :

```
typedef void CALLBACK (CallbackFunc) (int);
Void DLL_EXP CalcResult (CallbackFunc * callback)
   {
     -
     -
     -
     If (Callback)
        Callback (parameter);
   }
```

The use of DLL_EXP symbol tells the compiler whether the function is being exported or imported. This symbol is defined in the DLL's header like this :

```
# if defined (_DLL_)
   # define DLL_EXP_export
# else
   # define DLL_EXP_import
# endif
```

Usage of Callback Functions

- Callback functions can be used to provide event handling in languages that do not have built in events. For example, we can call a DLL function by passing a pointer to one of our functions, which will be called when the requisite time has passed.

6.3.5 Difference between Callback Function and Normal Function

- Normally, an application program calls functions in the DLL. Sometimes, however, the calling application needs periodic information from the DLL, while a processing is taking place. For example, if the DLL is doing some lengthy calculations, it would be useful for the calling application to receive

Scanned with CamScanner

periodic reports of the calculation, such as the percentage complete. The calling application could use this information to display a progress bar.

- Callback functions can be used to provide event handling in languages that do not have built in events.

Example 6.3.1

Explain the difference between .EXE and .DLL file.

Solution :

A DLL file is a collection of functions, which can be called upon when needed by the executable program (EXE) that is running.

A DLL is an executable file, that cannot run own its own, it can only run when called inside an executable file.

The EXE file is primarily used by Microsoft for executable file. An EXE file can be executed independently.

6.4 Dynamic Linking with and without Import

Here, an example is being given to show the creation of a basic DLL and how to use it (with import library linking and without).

If you are using vc++, create a new project and set the target as a win32 Dynamic_link library. This way when you compile your code, it should produce :

1. A dll
2. An import library (.lib)
3. An export library (.exp)

Here is a sample code that shows how the sample DLL is put together :

☞ **Header file (dlltest.h) :**

```
# ifndef _DLLTEST_H_
# define _DLLTEST_H_
# include <iostream.h>
# include <stdio.h>
# include <windows.h>
extern "C" _declspec (dllexport) void NumberList ();
extern "C" _declspec (dllexport) void LetterList ();

# endif
```

☞ **The Source file (dlltest.cpp)**

```
# include "dlltest.h"
# define MAXMODULE 50
char module [MAXMODULE];

extern "C" _declspec (dllexport)
void NumberList ()
{
   GetModuleFileName (NULL, (LPTSTR) module,
MAXMODULE);
   cout << "\n\nThis function was called from"
      << module
      << endl << endl ;
   Cout << "NumberList () : " ;

   for (int i = 0; i < 10; i++)
   {
     Cout << i << " " ;
   }
   cout << endl << endl ;
}
extern "C" declspec (dllexport)
void LetterList ()
{
   GetModuleFileName (NULL, (LPTSTR)module,
MAXMODULE);
   cout << "\n\nThis function was called from"
     << module
     << endl << endl ;
   cout << "LetterList () : " ;
   for (int i = 0; i < 26; i++)
   {
     cout << char (97 + i) << " " ;
   }
   cout << endl << endl;
}
```

The above code contains two simple functions. The first function "NumberList", displays values from 0 to 9 and also tell you where they are being called from. The extern "C" _declspec (dllexport) means that we want to allow these functions to be used by our actual program. When you compile this, it should create the libraries.

6.4.1 Using the DLL (with an import library)

Using DLL with the import library is the easiest. All that is required to be done is :

1. Include the header file
2. Include the import library when you are linking object files.

Here is an example :

☞ **DLL test source file (dllrunol.cpp)**

```
# include <conio.h>
# include <dlltest.h>

Void main ()
{
   NumberList ();
   LetterList ();

   getch ();
}
```

The code above will work fine if you have the followings :

1. dlltest.h header file in the compiler's header path.

Scanned with CamScanner

2. dlletest.lib import library in the lib path.
3. import library must be linked with other modules while linking.

dllrun01.exe output

```
This function was called from C: \
DALLTEST\DLLRUN01.EXE
NumberList() : 0 1 2 3 4 5 6 7 8 9
This function was called from C:\
DALLTEST\DLLRUN01.EXE
LetterList () : a b c d e f g h i j k l m n o p q r t u v w x y z
```

6.4.2 Using DLL (without an important library)

DLL can be loaded without an import library on the fly. This is useful in plug-in system.

Here is the example code :

☞ **Source file (dllrun02.cpp)**

```
#include <windows.h>
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#define MAXMODULE 50
typedf void (WINAPI*cfunf) ();
cfunc NumberList ;
cfunc LetterList ;
void main () {
   HINSANCE hLib=LoadLibrary ("DALLTEST.DLL") ;
    if(hLib==NULL){
      cout << "Unable to load library" << endl;
      getch ();
      return;
    }
    char mod [MAXMODULE];
    GetModuleFileName ((HMODULE) hLib, (LPTSTR)
mod, MAXMODULE);
    cout << "Library loaded : " << mod << endl ;
    NumberList=(cfunc) GetProcAddress ((HMODULE)
hLib, "NumberList");
    LetterList=(cfunc) GetProcAddress ((HMODULE)
hLib, "LetterList");
    if ((NumberList ==NULL))||(LetterList == NULL)
)
{
      cont << "unable to load function (s)." << endl ;
      FreeLibrary ((HMODULE) hLib) ;
      return ;
    }
    NumberList ();
    LetterList ();
    FreeLibrary ((HMODULE) hLib);
    getch () ;
}
```

- This code should load the dll library.
- It will be able to dynamically link with the two functions that we want to call.

Example 6.4.1

Comment on the statement "Static binding leads to more efficient execution of a program than dynamic bindings"

Solution : Static binding is created before execution of the program. It is carried out by the linkage editor which produces an executable file. Since, there is no need for linking before calling a function during run time, it is more efficient on the contrary, static binding requires more memory as linked routines are part of the executable program.

Example 6.4.2

Why are library routines usually relocatable ? What would happen if these routines are made non-relocatable ?

Solution :

Non-relocatable program must be loaded at a fixed place in the memory. This is a serious limitation in multiprogramming environment. In addition a swapped out process can be swapped in at a different memory location. If multiple programs use the same library routine then it will be impossible to run them together.

6.5 Exam Pack (University Questions)

☞ **Syllabus Topic : Relocation and Linking Concepts**

Q. Explain the need of a linker in program development. *(Refer section 6.1.1)* (4 Marks) **(May 2013)**

Q. What is linker ? *(Refer section 6.1.1)* (2 Marks) **(May 2014)**

Q. Explain the term linker. *(Refer section 6.1.1)* (4 Marks) **(Dec. 2014)**

Q. Why program relocation is required and how is it performed? *(Refer section 6.1.2)* (6 Marks) **(May 2013)**

☞ **Syllabus Topic : Self Relocating Programs**

Q. Explain the need of program relocatability. *(Refer section 6.2)* (4 Marks) **(Aug. 2015(In Sem))**

☞ **Syllabus Topic : Static and Dynamic Linker**

Q. Explain dynamic loading and linking. *(Refer section 6.3.2)* (4 Marks) **(Oct. 2016(In Sem))**

❑❑❑

Scanned with CamScanner

UNIT III

CHAPTER 7

Introduction to OS

Syllabus Topics

Architecture, Goals and Structures of O.S., Basic functions, Interaction of O.S. and hardware architecture, System calls, Batch, multiprogramming, Multitasking, time sharing, parallel, distributed and real-time O.S.

Syllabus Topic : Architecture, Goals

7.1 Evolution of O.S. Functions

➔ **(May 2014, Dec. 2014)**

Q. What is an operating system ?

SPPU - May 2014, 4 Marks

Q. Explain the term : Operating System.

SPPU - Dec. 2014, 2 Marks

An operating system is an interface between users and the hardware of a computer system. It is a system software which may be viewed as an organised collection of software consisting of procedures for operating a computer and providing an environment for execution of programs.

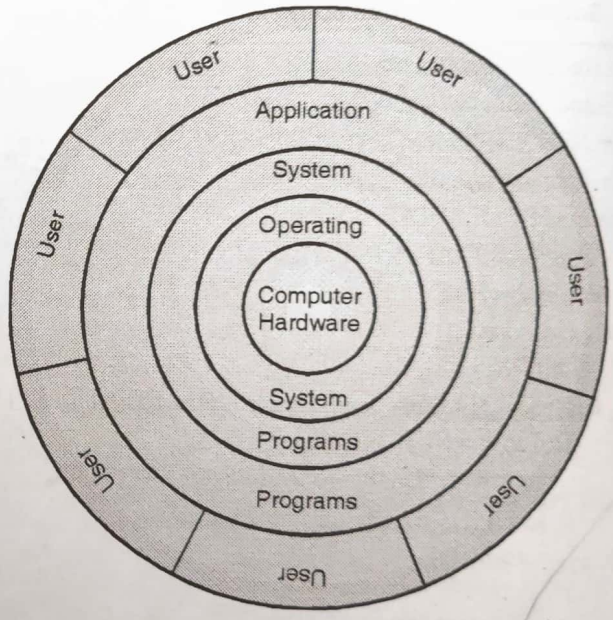
Operating system functions have evolved in response to the following considerations :

1. Efficient utilization of computing resources.
2. New features in computer architecture.
3. New user requirements.

An operating system manages resources of a computer. These resources include :

1. Memory
2. Processor
3. File system.
4. Input/output devices.

It keeps track of the status of each resource and decides who will have a control over computer resources, for how long and when. The positioning of an operating system in overall computer system is shown in the Fig. 7.1.1.



(S7.1) **Fig. 7.1.1 : Components of a computer system**

- Operating system controls computer hardware resources.
- Programs interact with computer hardware with the help of operating system.
- There are two ways in which one can interact with operating system :
  1. By making a system call.
  2. By operating system commands.

Syllabus Topic : Basic Functions, Interaction of O.S. and Hardware Architecture

7.1.1 Functions of an Operating System

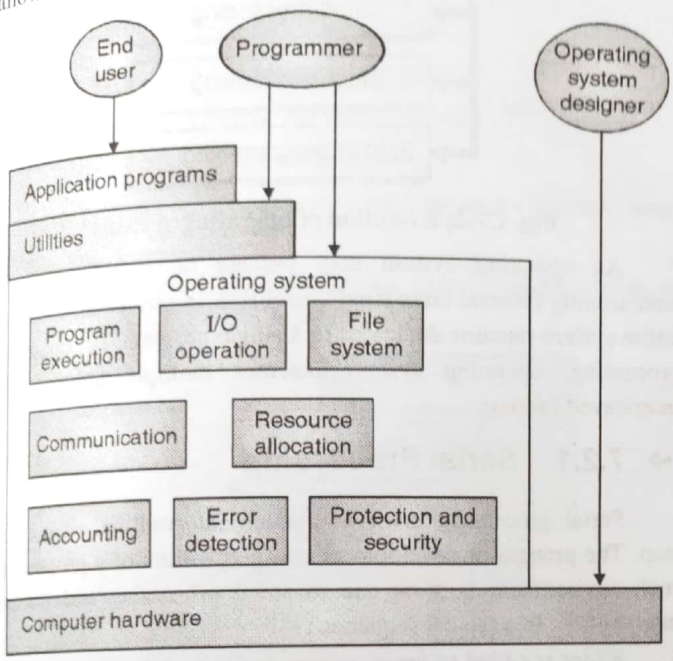
➔ **(May 2013, Dec. 2013, Dec. 2016)**

Q. Explain functions of an Operating System.

SPPU - May 2013, Dec. 2013, Dec. 2016, 8 Marks

Scanned with CamScanner

An operating system provides an interface between the user and the hardware. It makes a computer more convenient to use. In addition, an operating system manages computer resources. It allows computer resources to be used in an efficient way.



(S6.3) **Fig. 7.1.2 : Layers and view of operating system services**

7.1.1(a) The Operating System as a User/Computer Interface

The operating system hides the details of the hardware from the user and makes it convenient to use the system. It works as an interface between user and the computer. Briefly, the operating system provides services in the following areas :

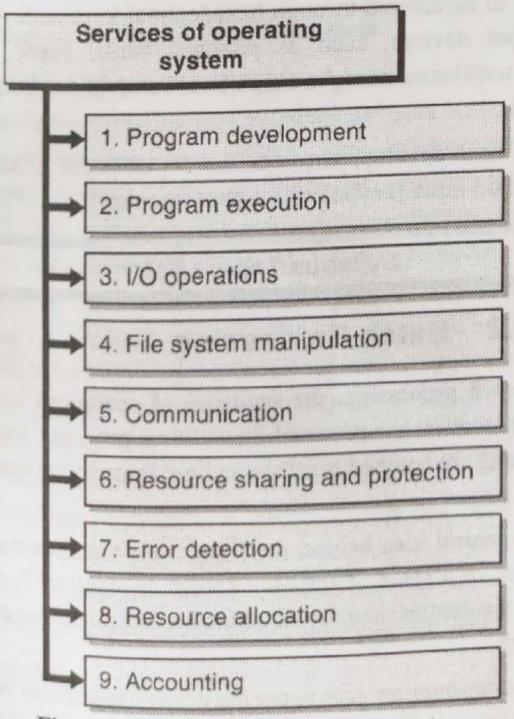


Fig. C7.1: Services of operating system

1. Program development

Operating system provides a set of utility programs which are necessary for program development. These utilities include :

1. Editor
2. Debugger
3. Compiler
4. Interpreter
5. Linker
6. Loader

2. Program execution

Operating system handles loading and scheduling of programs. Several activities are involved in program execution :

1. Loading of instructions and data
2. Initialization of files and I/O devices
3. Allocation of other resources.

3. I/O operations

A running program may require I/O. Users are not allowed to control I/O devices directly. Operating system provides a uniform interface and hides details of I/O devices from the user. Therefore an operating system must provide means for doing I/O.

4. File system manipulation

An operating system provides various system calls for manipulation of files. Details of the underlying secondary storage remain hidden from the user. In addition, in a multiuser environment, the operating system must provide protection mechanism to control access to the files.

5. Communication

Under many situations a process has to communicate with another process. Communication is required for exchange of information. Such communications may occur between two processes running on the same computer or different computers. The communication between two processes is implemented by the operating system through the following techniques :

1. Shared memory
2. Pipe
3. Message passing

6. Resource sharing and protection

Sharing and protection of resources allocated to a process is provided by the operating system. In a multiprogramming a process is not allowed to interfere with other processes. Protection involves ensuring that all accesses to system resources are controlled.

7. Error detection

Different types of errors may occur while a computer system is running. These errors could be due to memory error, connection failure, error in I/O devices etc. For each type of error, the operating system should take appropriate action to ensure consistent computing.

8. Resource allocation

Computer resources are managed by the operating system. These resources include :

1. Processor
2. Memory
3. File storage
4. I/O devices

When multiple users run their programs, resources must be allocated to them.

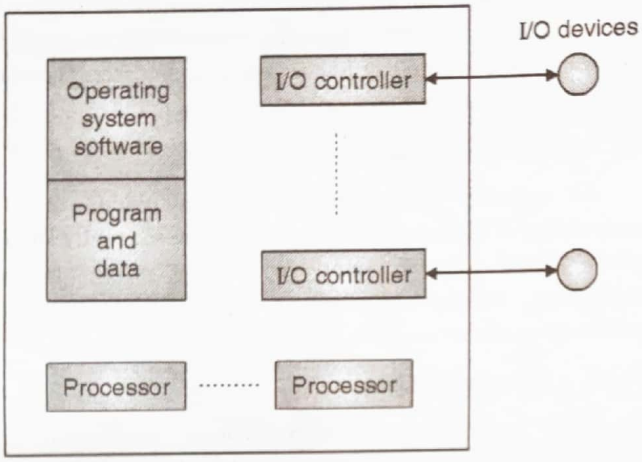
Scanned with CamScanner

9. Accounting

Each operating system is required to keep track of which users use how much and what kinds of computer resources. Usage statistics are useful in fine tuning computer services.

7.1.1(b) The Operating System as Resource Manager

The operating system is responsible for resource management. Resources are required for storing and running of programs. Resources managed by an operating system are shown in the Fig. 7.1.3.



(S6.4)**Fig. 7.1.3 : The operating system as resource manager**

The major resources managed by an operating system include :

1. Memory
2. Processor
3. I/O devices

When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them.

- The operating system itself is a program that directs the processor in the use of the other resources and in the timing of its execution of other programs.
- In-between, the operating system relinquishes the control of CPU for program execution.
- A small portion of the operating system, known as kernel always resides in the main memory.
- The allocation of main memory is controlled jointly by the operating system and the memory management hardware in the processor.
- The operating system decides when an I/O device can be used by a particular process. It also controls file system.
- The operating system decides about allocation of processor to a particular program.

7.2 Evolution of Operating Systems

(Dec. 2016)

Q. List different types of operating systems with examples.
SPPU - Dec. 2016, 3 Marks

Operating system has evolved over a period of time. In particular, these are :

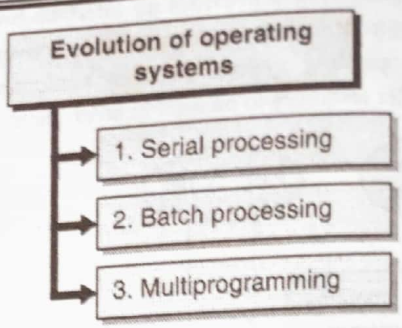


Fig. C7.2: Evolution of operating systems

An operating system may process its tasks serially or concurrently (several tasks simultaneously). In serial execution, the entire system remains dedicated to a single program. In concurrent processing, operating system executes multiple programs in interleaved fashion.

7.2.1 Serial Processing

Serial processing involves manual intervention after every step. The process of development and preparation of a program in such environment is slow due to serial processing and manual intervention. In a typical sequence :

1. Editor is called to create source code of user program written in a programming language.
2. Compiler is called to convert a source code into binary code.
3. Loader is called to load the executable program into main memory for execution.

The serial mode of execution is not efficient. Time is wasted after every step due to manual intervention. This results in low utilization of resources. In early computer systems, instructions and data used to be entered through hexadecimal keyboard. Advent of input/output devices, such as punched cards, paper tape and language translators brought a significant step in computer system utilization.

The next development was the replacement of card-decks with standard input (keyboard) / output (monitor).

Syllabus Topic : Batch

7.2.2 Batch Processing

In batch processing, the sequence of operations involved in program execution is automated. In addition, programs with similar requirements are batched together and run through the computer as a group.

The central idea behind the simple batch processing scheme was the use of a piece of software known as monitor. The monitor is always resident in memory. It automatically sequences one job to another job.

- Resident monitor acts as per the control statements given by a programmer.
- The control statements are for :
  1. Compiling a program
  2. Loading a program
  3. Executing a program

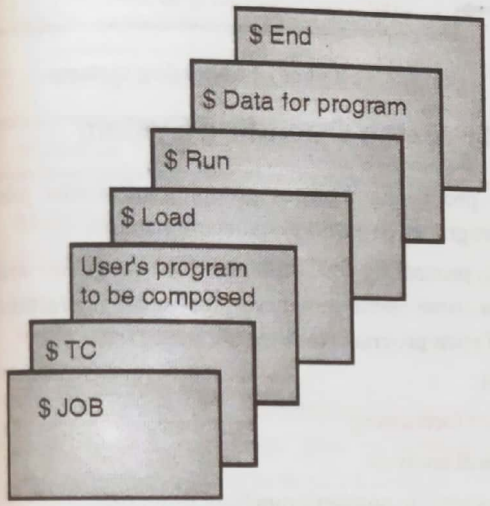
Scanned with CamScanner

- Control statements are specified using job control language. An example of job control language is given below :

  $ JOB – First card of a job
  $ TC – Execute the C-compiler
  $ LOAD – Load program into memory
  $ RUN – Execute the user program
  $ END – Last card of the job

Fig. 7.2.1 shows a sample card deck set up for a simple batch system.

- Batch processing is advantageous over serial processing as there is no intervening idle time.



(S7.2)**Fig. 7.2.1 : Card deck for a C-Program for a simple batch system**

☞ Buffering of Input/Output devices

The throughput of a computer system can be improved through buffering. Input/output activities are very slow. The speed discrepancy between fast CPU and comparatively slow input/output devices such as keyboard, printer is a major bottleneck. The problem is that while an input/output is occurring, the CPU is idle, waiting for the input/output to complete.

Buffering is a method of overlapping input, output and processing of a single job. The idea behind buffering is very simple. Each device has its own local memory.

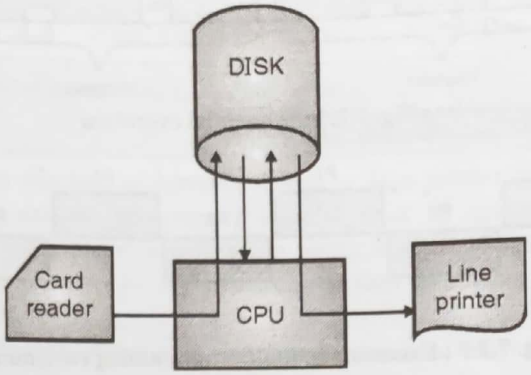
- During output operation, the CPU writes the data in device buffer and continues with further execution. It does not wait for the completion of input/output operation.
- During input operation, the input device can fetch and store the data in buffer in advance.

Thus, buffering can improve system throughput by overlapping input, output and processing of a single job.

☞ Spooling

If the CPU is, on the average much faster than an input device, buffering will be of little use.

If the CPU is always faster, then it always finds an empty buffer and have to wait for the input device. For example, the CPU can proceed at full speed until, eventually all system buffers are full. Then the CPU must wait for the output device. This situation occurs with input/output bound jobs where the amount of input/output activity is much more compared to computation.



(S7.3)**Fig. 7.2.2 : Spooling**

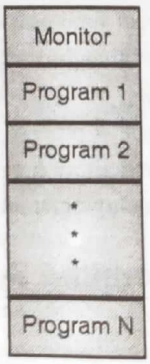
SPOOLING (simultaneous peripheral operation on line) is a more sophisticated form of input/output buffering. It uses the disk as a very large buffer for reading and for storing output files. Spooling allows CPU to overlap the input of one job with the computation and output of other jobs. Therefore, this approach is faster than buffering. Even in a simple system, the spooler may be reading the input of one job while printing the output of a different job.

Syllabus Topic : Multiprogramming

➜ 7.2.3 Multiprogramming

Multiprogramming increases resource utilization. When the current job waits for input/output operation to complete, the CPU can take up next job for execution. A single user cannot always keep CPU or input/output devices busy at all times. In order to increase the resource utilization, systems supporting multiprogramming approach allow more than one job (program) to utilize CPU time at any moment. More the number of programs competing for system resources, the better will be resource utilization.

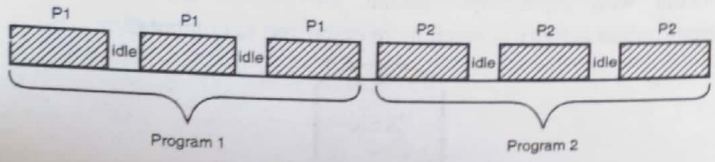
In multiprogramming, the main memory of a system contains a number of programs (Fig. 7.2.3).



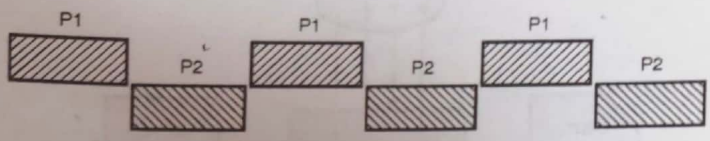
(S7.4)**Fig. 7.2.3 : A multiprogramming system with N jobs**

The operating system takes one of the programs and start executing. Let us consider a system executing two programs. During execution of program 1, it may require input/output operation. In sequential execution (Fig. 7.2.4), the CPU will sit idle. In multiprogramming system (Fig. 7.2.5), the operating system will simply switch over to the next program.

Scanned with CamScanner



(S7.5)**Fig. 7.2.4 : Sequential execution**



(S7.6)**Fig. 7.2.5 : Execution in multiprogramming environment**

- Multiprogramming increases resource utilization.
- It can support multiple simultaneously interactive users (terminals).
- Compare to sequential operating system, multi-programming operating systems are complex.
- Memory sharing, memory protection, synchronization are some of the important issues in multi-programming environment.

Example 7.2.1

Justify the statement : "It is possible to support multiprogramming without using time sharing. However, it is impractical to support time sharing without using multiprogramming.

Solution :

- Multiprogramming implies multitasking. In multiprogramming, instructions and data from two or more separate processes are required to reside in primary memory simultaneously. The simplest form of multiprogramming can be implemented through serial multitasking or context switching. It is nothing more than stopping one process temporarily to work on another. Thus, it is possible to support multiprogramming without using time sharing.
- Time sharing is a special case of multiprogramming, where a CPU serves a number of processes by switching context at regular intervals. Most time sharing system use time-slice (round robin) scheduling of CPU. When the CPU switches from one process to another process, the new process should be in the memory. Thus, it is impractical to support time sharing without using multiprogramming.

7.3 Types of Operating System

➔ **(May 2014)**

Q. List the various types of the O.S. with their basic functions. **SPPU - May 2014, 4 Marks**

There are different types of operating systems. These include :

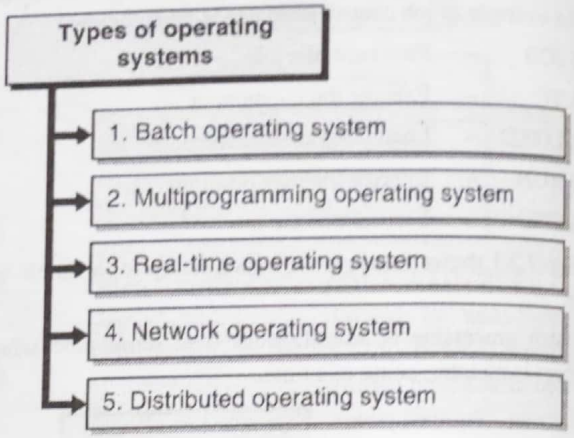


Fig. C7.3 : Types of operating systems

➔ 7.3.1 Batch Operating System

Batch processing requires grouping of similar jobs which consist of programs, data and system commands.

Batch processing is suitable for program with large computation time with no need for user interaction. Some examples of such programs include :

1. Payroll
2. Weather forecasting
3. Statistical analysis
4. Large scientific number crunching programs

Users are not required to wait for the completion of programs. Program execution may need few hours to few days for execution.

Batch processing has two major disadvantages :

1. Non-interactive environment
2. Off-line debugging

A batch operating system allows practically no interaction between users and executing programs. The turnaround time taken between job submission and job completion in batch operating system is very high. Users have no control over intermediate results of a program.

In batch operating system, a programmer cannot debug program bugs the moment it occurs.

- Process scheduling, memory management, file management and input/output management in batch processing are quite simple.
- Jobs are typically processed in the order of submission.
- Memory is usually divided into two areas :
  1. One portion contains the operating system.
  2. Second part is for executing the user program.
- When the execution of one program is over, the new program is loaded into the same area.
- Allocation of input/output devices is simple as there is only one program in execution at a time.
- Access to files is serial and there is hardly a need of protection and file access control mechanism.

Scanned with CamScanner

7.3.2 Multiprogramming Operating System

There are various forms of multiprogramming including :

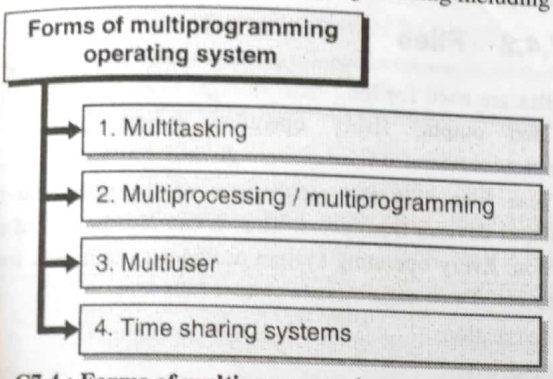


Fig. C7.4 : Forms of multiprogramming operating system

Syllabus Topic : Multitasking, Parallel

1. Multitasking Operating Systems

A multitasking operating system supports two or more active processes simultaneously. It allows the instructions and data from two or more processes to reside in primary memory simultaneously.

The simplest form of multitasking is based on serial multitasking or context switching. This is nothing more than stopping one process temporarily to work on another.

2. Multiprocessing / Multiprogramming

In multiprocessing, multiple CPUs perform more than one job at a time. The term multiprogramming refers to the situation in which a single CPU divides its time between more than one job. Time sharing is a special case of multiprogramming, where a single CPU serves a number of users at interactive terminals.

3. Multiuser Operating System

Multiuser operating system allows simultaneous access to a computer system through two or more terminals. A dedicated railway reservation system supports hundreds of terminals in an example of multiuser operating system.

Syllabus Topic : Time Sharing

4. Time Sharing System

It is a form of multi-programmed operating system which operates in an interactive mode with a quick response time. A time sharing system allows many users to simultaneously share the computer resources. Since each action or user command requires very little CPU time, the CPU switches quickly from one user to another user.

Most time sharing systems use time-slice (round robin) scheduling of CPU.

- Memory management in time sharing provides for protection and separation of user programs.
- Operating system should be able to handle multiple users.

Multiprogramming operating systems are quite complicated. It provides high system throughput and resource utilization.

Syllabus Topic : Real Time O.S.

7.3.3 Real-time System (Dec. 2015)

Q. What is real time operating system?

SPPU- Dec. 2015, 3 Marks

In this form of operating system, a large number of events mostly external to computer systems, must be accepted and processed in a short time or within certain deadlines. Examples of such applications are flight control, real time simulation, process industry etc.

- Primary objective of real-time system is to provide quick response time. Resource utilisation is of secondary concern in a real-time system.
- In real-time system there is a little swapping of programs between primary and secondary memory. Most of the time, processes remain in primary memory in order to provide quick response.
- Time-critical device management is one of the main characteristics of real-time system. It also provides sophisticated form of interrupt management and input/output buffering.
- The primary objective of file management in real-time systems is usually the speed of access rather than efficient utilization of secondary storage.

7.3.4 Network Operating System

A network operating system is a collection of software and associated protocols that allow a set of autonomous computers which are interconnected by a computer network to be used together in a convenient and cost effective manner. In a network operating system, the users are aware of existence of multiple computers and can log in to remote machines and copy files from one machine to another machine.

Some typical network operating systems have following features :

1. Each computer has its own private operating system.
2. Each user normally works on his own system but he can access other systems on the network.
3. It allows users to access the various resources of the network hosts.
4. It allows controlling of access so that only users with proper authorisation are allowed to access particular resources.
5. It makes remote resources appear to be identical to local resources.
6. It provides up-to-the minute network documentation on-line.

Syllabus Topic : Distributed O.S.

7.3.5 Distributed Operating System

A distributed operating system is one that looks to its users like an ordinary centralized operating system but runs on multiple independent CPUs.

Scanned with CamScanner

- User sees the environment as a virtual uniprocessor environment and not as a collection of distinct machines.
- In distributed systems, users are not aware of where their programs are being run or where their files are residing.
- Distributed operating system allows programs to run on several processors at the same time without the users being aware of this distribution.
- Distributed systems are more reliable than uniprocessor based systems. They continue to work even if some portion of the system is malfunctioning.
- The distributed systems have a price/performance advantages over traditional centralized systems.
- The distributed systems support incremental growth. New computer systems can be added to the network as per the processing needs.
- Distributed environment provides high reliability and availability.

7.4 Operating System Components

Some of the important operating system concepts include :

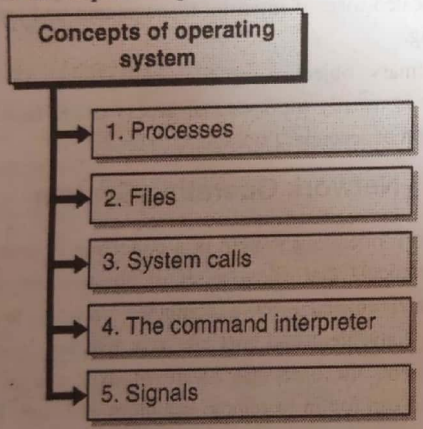


Fig. C7.5 : Concepts of operating system

→ 7.4.1 Processes

A process is program in execution. It consists of the followings :

1. Executable program
2. Program's data
3. Stack and stack pointer
4. Program counter and other CPU registers
5. Details of opened files

- A process can be suspended temporarily and the execution of another process can be taken up. A suspended process can be re-started at a later time.
- Before suspending a process, its details are saved in a table called the process table.
- An operating system supports two system calls to manage processes :
  1. **Create** to create a new process.
  2. **Kill** to delete an existing process.
- A process can create a number of **child processes**.
- Process can communicate among themselves either using shared memory or by message passing techniques. Two processes running on two different computers can communicate by sending messages over a network.

→ 7.4.2 Files

Files are used for long term storage. Files are used both for input and output. Every operating system provides file management service.

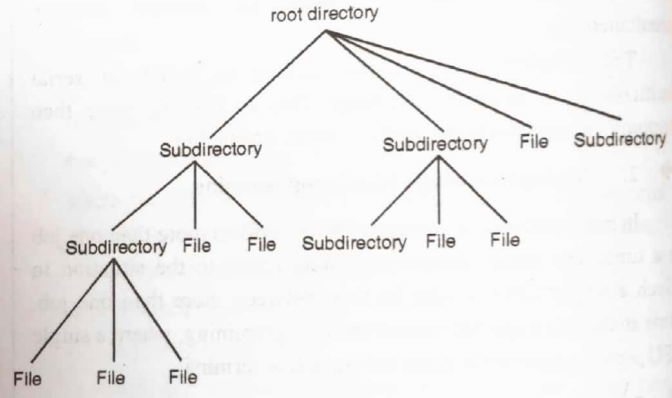
These files hide the peculiarities of the disks and other input/output devices from users. Thus it can also be treated as an abstraction. Every operating system provides system calls for file management. These calls include system calls for :

1. File creation
2. File deletion
3. Read and write operations

Files are stored in directory. System calls are provided :

1. To put a file in a directory
2. To remove a file from a directory

The directory entry may be a file or a directory itself. Directory structure is a hierarchical structure (Shown in Fig. 7.4.1)

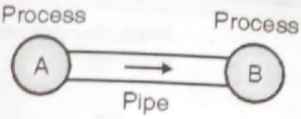


(S7.7)Fig. 7.4.1 : Tree-structured directory

- Every file can be specified by giving its path from the root.
- At every instant, each process has a current working directory.
- Normally, files in a system are protected so that the privacy of each person's files can be maintained.
- Many operating systems represent each input/output device as a special file. This makes input/output devices look like a file. There are two kinds of special files :
  1. Block special files.
  2. Character special files.

  Block special files can model a device containing a set of randomly accessible blocks. Character special files can model a device consisting of character streams like monitor, keyword etc.
- Many operating systems support a special file known as pipe. Pipe is a Pseudo file that can be used to connector two processes. Output of one process can be sent as input to another process through a pipe. It is shown in Fig. 7.4.2.

Scanned with CamScanner



(S7.8) Fig. 7.4.2 : Two processes connected by a pipe

Syllabus Topic : System Calls

7.4.3 System Calls

(Dec. 2013, Dec. 2014, May 2015, Dec. 2015, May 2016)

Q. What is purpose of system calls in an operating system ? List types of system calls and explain any one of them. **SPPU - Dec. 2013, 8 Marks**

Q. List the categories of system calls and explain process system call with an example. **SPPU - Dec. 2014, May 2015, 6 Marks**

Q. Write short notes on : System Call. **SPPU - Dec. 2015, 3 Marks**

Q. List the different categories of system calls and explain in brief any two of them. **SPPU - May 2016, 6 Marks**

System calls provide an interface to the services made available by an operating system. User program interact with operating system through system calls.

- These calls are normally available as library functions in a high level language such as C.
- In an assembly language, necessary parameters are stored in specified CPU-registers and then it issues a software interrupt instruction. System calls in DOS are availed by using "INT 2IH".
- Execution of the following instruction to read n bytes from a file :

  Count = read (filepointer, buffer, n);

  will internally make a system call to read n bytes from the specified file and the data will be stored in the array buffer.
- The number and type of system calls varies from operating system to operating system.

In general, system calls are available for the following operations :

1. Process management
2. Memory management
3. File operations
4. Input/Output operations

Advantages of system calls

- Most of the programming languages have a built-in library. These library functions work as the link to system calls made available by the operating system.
- It provides a level of abstraction. The caller need know nothing about how the system call is implemented or what it does during execution. Details of the operating-system interface are hidden from the programmer.
- Different hardware services can be availed through system calls.

7.4.4 Command Interpreter

There are several ways for users to interface with the operating system. One of the approaches of user interaction with operating system is through commands.

Command interpreter provides command-line interface. It allows user to enter a command on command line prompt. The command interpreter accepts and executes a command entered by an user.

- For example, shell is a command interpreter under UNIX. A $ at the beginning of the command line tells the user that the shell is waiting to accept a command. User can display the contents of the file file1.txt using the following command :

  $ cat file1.txt
- The commands to be executed are implemented in two ways :
  1. Command interpreter itself contains the necessary code to be executed.
  2. Command is implemented through a system file. To execute a command the necessary system file is loaded into memory and executed.

7.4.5 Signals

Signals are used in operating system to notify a process that a particular event has occurred. A signal follows the following pattern :

1. A signal is generated by the occurrence of a particular event. The event can be clicking of mouse, divide by zero operation in a program etc.
2. A generated signal is delivered to a process.
3. Once delivered, the signal must be handled.

- A signal can be both synchronous and asynchronous.
- Every type of signal can be handled by the operating system. Operating system has a default signal handler. A user can also define his own signal handler.
- The signal causes the process to temporarily suspend whatever it was doing, save its registers on the stack, and start running a special signal handling procedure.
- Signals are the software analog of hardware interrupts. Many illegal conditions detected by hardware during program execution are also converted into signals to the guilty process. Signals are also used for inter-process communication.

Example 7.4.1

What is the difference between a process and a program ?

Solution : A program is a passive entity. A process is basically a program in execution. A program by itself is not a process. A program is a passive entity, like the contents of a file stored on disk. A process is an active entity. A process consists of the followings :

1. Executable program
2. Program's data
3. Stack and stack pointer
4. Program counter and other CPU registers.
5. Details of opened files.

Scanned with CamScanner

A process can be suspended temporarily and the execution of another process can be taken up. A suspended process can be re-started at a later time.

Example 7.4.2

State whether the following statements are TRUE or FALSE :

(i) UNIX is the multiprogramming O.S.

(ii) Short term scheduler is responsible for ready to running state transition.

(iii) Process is the passive entity.

(iv) Process concept is used by Batch O.S

Solution :

(i) True (ii) True

(iii) False (iv) False

Example 7.4.3

Define the following terms :

(i) Job (ii) Step (iii) Process

(iv) Privileged instruction (v) Multiprogramming

(vi) Real time operating system.

Solution :

(i) **Job :** A job may consist of a program or a set of programs. These programs are required to be executed in a pre-defined sequence for any application.

(ii) **Steps :** Execution of a program requires several steps like :

1. Compilation
2. Linking
3. Loading and execution.

(iii) **Process :** Refer Section 7.4.1.

(iv) **Privileged instructions :** Privileged instructions are meant for operating system. These instructions are executed in kernel mode.

(v) **Multiprogramming :** Refer Section 7.2.3.

(vi) **Real time operating system :** Refer Section 7.3.3.

Example 7.4.4 :

State True or False

(i) Time sharing operating system allows number of users to execute their programs on the same machine by sharing time among the users.

(ii) MS-DOS is single user operating system.

(iii) Multiprogramming increases the CPU utilization.

(iv) Multiprocessing is a feature of operating system where many processors are used and controlled by one operating system.

Solution :

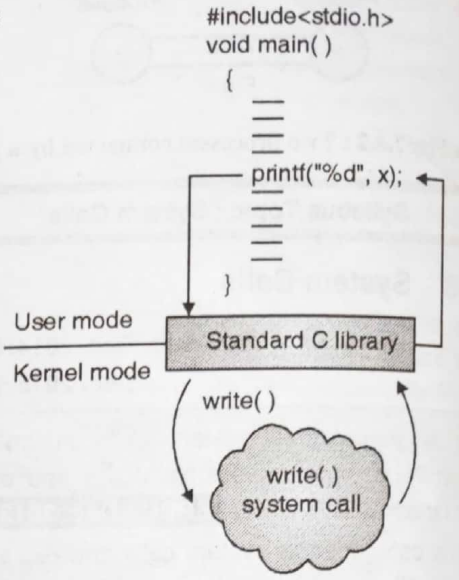
(i) True (ii) True

(iii) True (iv) False

Example 7.4.5

Using simple system call as an example describe the steps generally involved in providing the result for the point of calling the function in C library to the where that function returns.

Solution :



(S6.5)**Fig. Ex. 7.4.5 : Handling by C library**

Many system calls are provided by the C-library.

- As shown in Fig. Ex. 7.4.5, the C-program invokes the printf() statement.
- In return, the C-library invokes the necessary system call in the operating system – the write() system call in this case.
- The C-library takes the value returned by write() and passes it back to the user program.

Syllabus Topic : Structure of O.S.

7.5 Operating System Structure

Operating system is a very large and complex software, supporting a large number of functions. It is developed as a collection of smaller modules. There are different operating system structures :

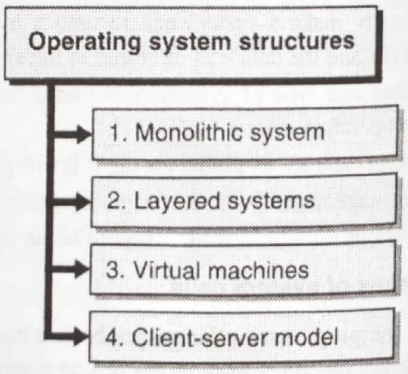
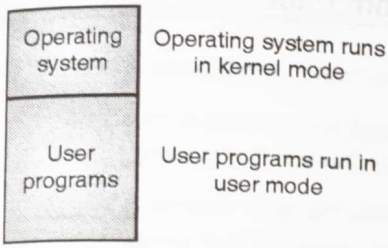


Fig. C7.6 : Operating system structures

➜ 7.5.1 Monolithic System

A monolithic operating system is written as a collection of procedures. Each of these procedures can call any other procedure. The operating system is written as a simple program and resides in one portion of the memory (Fig. 7.5.1).

Scanned with CamScanner



(S7.9)**Fig. 7.5.1 : A monolithic system**

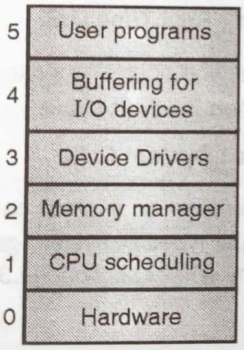
In a monolithic system, services are provided through system calls. A system call switches the machine from user mode to kernel mode. Kernel mode is also known as supervisory mode. Privileged instructions can be executed in supervisory mode.

Each system call has a corresponding service procedure.

Monolithic system cannot be used for a complicated operating system. Such an operating system is best suited for single user environment without multi-programming.

➜ 7.5.2 Layered Systems

The operating system architecture based on layered approach consists of number of layers, each built on top of lower layer. A typical layered architecture is shown in Fig. 7.5.2.



(S7.10)**Fig. 7.5.2 : A typical layered operating system**

- Layer 0 deals with hardware.
- Layer 1 deals with allocation of CPU to processes.
- Layer 2 implements memory management such as Paging, segmentation etc.
- Layer 3 deals with device drivers. Device drivers provide device independent interaction with devices.
- Layer 4 deals with input/output buffering.
- The highest layer 5 deals with user interfaces.

☞ Advantages of layered approach

1. Layered approach provides modularity. It helps in debugging and verification of the system.
2. Higher layer utilizes services provided by lower layers. A higher layer need not know how these operations are implemented only what these operations do.
3. Any layer can be written and debugged without any concern about the rest of the layers.

☞ Disadvantages of layered approach

1. The major difficulty with the layered approach is how to differentiate one layer from another.
2. Since a layer can use services of a layer below it, it should be designed carefully. For example, the device driver for secondary memory must be at a lower level than memory management routine.

➜ 7.5.3 Virtual Machine

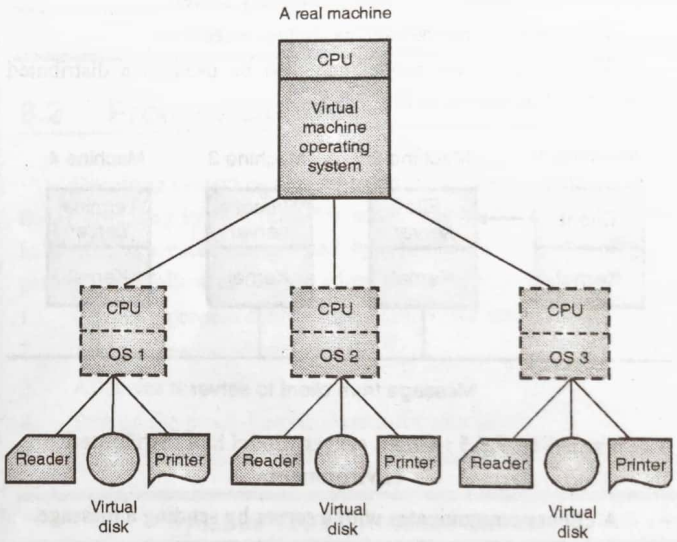
It is a concept which creates an illusion of a real machine. It is created by a virtual machine operating system that makes a single real machine appear to be several real machines.

An important aspect of this technique is that each user can run operating system of his choice. This concept is shown in Fig. 7.5.3.

In a virtual machine environment, a single real machine gives an illusion of several virtual machines, each having its own :

1. Virtual processor
2. Virtual storage and input/output devices

Process scheduling can be used to share the CPU and make it appear that each user has his own processor.



(S7.11)**Fig. 7.5.3 : Creation of several virtual machines by a single physical machine**

☞ Advantages of virtual machine

1. Concurrent running of dissimilar operating system by different users.
2. No need for conversion of a program written to run under a specific operating system.
3. Programs can be developed and debugged for machines different from the host machine.
4. The high degree of separation between independent virtual machines helps in ensuring privacy and security.

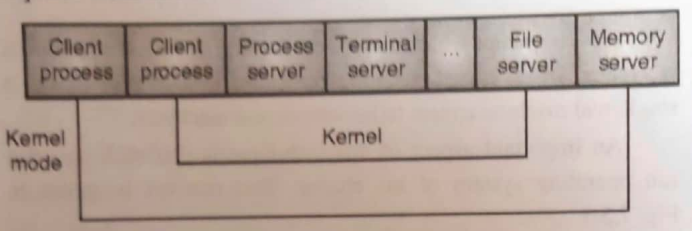
The heart of such systems is virtual machine monitor which runs on the bare hardware. The virtual machine monitor provides virtual environment.

➜ 7.5.4 Client-Server Model

Client-Server computing has become very popular. It has been replacing both main-frame systems and other computing environments based on centralized approach. In a client-server

Scanned with CamScanner

operating system the kernel part (core of the operating system) is very small. Most of the operating system functions are implemented using user processes. In this model, to request a service, such as reading a file, a user process sends a request to a server process (shown in Fig. 7.5.4), the server process does the required work and sends back the answer.



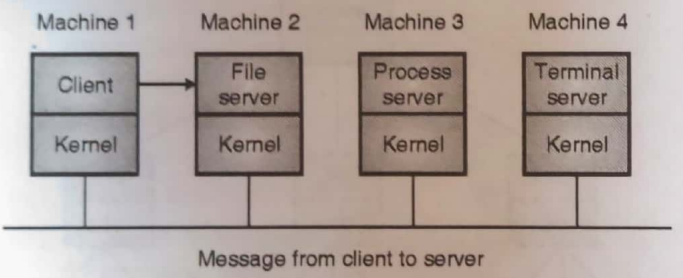
(S7.12)**Fig. 7.5.4 : The client-server model**

In this model, the kernel handles the communication between clients and servers. Independent server processes handle different aspects of operating systems. These processes are :

1. File server.
2. Process server.
3. Terminal server.
4. Memory server.

These servers run as user-mode processes.

The client server model can also be used in a distributed environment as shown in Fig. 7.5.5



(S7.13)**Fig. 7.5.5 : Client server model in a distributed environment**

- A client communicates with a server by sending a message.
- A client server model provides :
  1. High scalability.
  2. Reliability
  3. Fault tolerance.
  4. Security.
  5. Reduces load on network.

7.6 Exam Pack (University Questions)

☞ **Syllabus Topic : Architecture, Goals**

Q. What is an operating system ? *(Refer section 7.1)* (4 Marks) **(May 2014)**

Q. Explain the term : Operating System. *(Refer section 7.1)* (2 Marks) **(Dec. 2014)**

☞ **Syllabus Topic : Basic Functions, Interaction of O.S. and Hardware Architecture**

Q. Explain functions of an Operating System. *(Refer section 7.1.1)* (8 Marks) **(May 2013, Dec. 2013, Dec. 2016)**

Q. List different types of operating systems with examples. *(Refer section 7.2)* (3 Marks) **(Dec. 2016)**

Q. List the various types of the O.S. with their basic functions. *(Refer section 7.3)* (4 Marks) **(May 2014)**

☞ **Syllabus Topic : Real Time O.S.**

Q. What is real time operating system? *(Refer section 7.3.3)* (3 Marks) **(Dec. 2015)**

☞ **Syllabus Topic : System Calls**

Q. What is purpose of system calls in an operating system ? List types of system calls and explain any one of them. *(Refer section 7.4.3)* (8 Marks) **(Dec. 2013)**

Q. List the categories of system calls and explain process system call with an example. *(Refer section 7.4.3)* (6 Marks) **(Dec. 2014, May 2015)**

Q. Write short notes on : System Call. *(Refer section 7.4.3)* (3 Marks) **(Dec. 2015)**

Q. List the different categories of system calls and explain in brief any two of them. *(Refer section 7.4.3)* (6 Marks) **(May 2016)**

❑❑❑

Scanned with CamScanner

CHAPTER 8

UNIT III

Process Management

Syllabus Topics

Process Concept, Process states, Process control, Threads, Scheduling : Types of scheduling : Preemptive, Non preemptive, Scheduling algorithms : FCFS, SJF, RR.

Syllabus Topic : Process Concept

8.1 Process

➔ (Dec. 2015)

Q. Explain process in detail. **SPPU - Dec. 2015, 3 Marks**

A process is basically a program while it is being executed. A process is more than program code. It also includes the followings :

1. Value of program counter.
2. Contents of CPU register.
3. Process stacks containing temporary data (such as function parameters, return address, and local variables).
4. Data section containing global variables.
5. Program code
6. Dynamically allocated memory during runtime.

A program in itself is not a process. A program is a passive entity, whereas a process is an active entity. A program becomes a process when an executable program is loaded into memory.

In a time shared system, several users programs reside in computer's memory. Each user is allocated only a fraction of CPU time (time slab) at a time for his program. When CPU executes a program or command, a process is created. When one process stops running because it has taken its share of CPU time, another process starts.

When a process is temporarily suspended, information about it, is stored in memory so that its execution could start from the same location where from it was suspended.

In many operating systems, all the information about each process is stored in a process table. Operating system also provides several system calls to manage processes. These system calls are mainly to create and kill processes.

Syllabus Topic : Process Control

8.2 Process Control

Operating system creates a process to execute a user program. Every operating system provides some way of creating a process. In UNIX, processes are created by the FORK system call. O.S. performs the following actions when a new process is created :

1. Creates a process control block (PCB) for the process.
2. Assigns process id and priority.
3. Allocates memory.
4. Sets up the process environment for execution.
5. Initializes resource accounting information for the process.

Syllabus Topic : Process States

8.2.1 Process States

➔ (May 2013, Dec. 2014, May 2015, May 2016, Dec. 2016, May 2017)

Q. Draw and explain process state transitions. **SPPU - May 2013, 6 Marks**

Q. Explain various states of a process with diagram. **SPPU - Dec. 2014, May 2015, May 2016, Dec. 2016, 6 Marks**

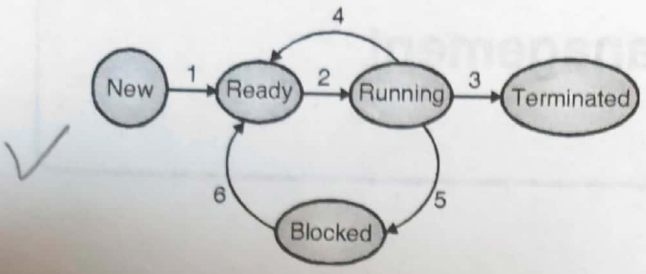
Q. What are the various states of processes and how it is managed by operating system ? **SPPU - May 2017, 6 Marks**

The lifetime of a process can be divided into several stages as states, each with certain characteristics that describe the process. It means that as a process starts executing, it goes through one state to another state. Each process may be in one of the following states:

Scanned with CamScanner

1. New
2. Ready
3. Running
4. Blocked
5. Terminated.

A general form of process state diagram is shown in Fig, 8.2.1.



(1) Admit **(4) Time slab over preemptive**
(2) Dispatch for execution **(5) Blocked for I/O**
(3) Release **(6) I/O activity is over**

(S8.1)**Fig. 8.2.1 : Process state transition diagram**

1. New

The process has been created.

2. Ready

The process is waiting to be allocated to a processor. Process comes to this state immediately after it is created. All ready processes are kept in a queue. A program called scheduler, which is a part of the operating system, picks up one ady process for execution by passing control to it.

3. Running

A CPU is currently allocated to the process and the process is getting executed.

- A running process is blocked if there is an I/O request during execution.
- A running process is sent to the ready queue after the time slab is over. After the time slab, the scheduler may schedule another process from the ready queue for execution.

4. Blocked

The process is waiting for completion of some I/O activity. Such processes are normally not considered for execution until the related blocking condition is fulfilled. The running process becomes blocked by invoking I/O routines, whose result is needed during execution.

5. Terminated : The process has finished its execution.

8.2.2 Process Implementation

→ (May 2013, Dec. 2016)

Q. Write short notes on : Process Control Block.
SPPU - May 2013, 3 Marks

Q. Draw and explain process control block.
SPPU - Dec. 2016, 6 Marks

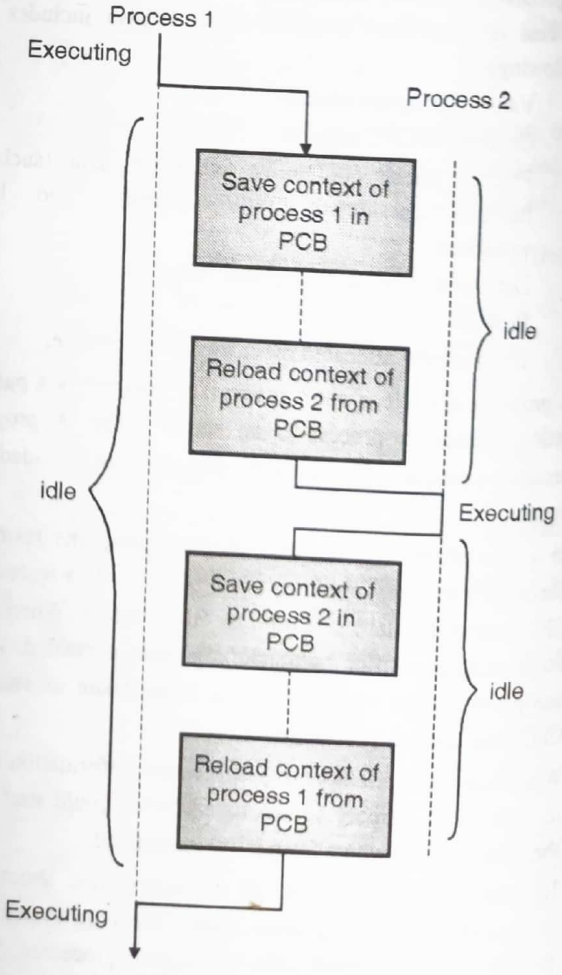
A data structure called the process control block (PCB) is used by the operating system to keep track of all information concerning a process. The operating system maintains a table of process control blocks with one entry per process. The process control block contains many pieces of information associated with a specific process, including :

1. **Process number :** Each process is identified by its process number, called process ID.
2. Priority.
3. **Process state :** Each process may be in any of these states :
   (a) New (b) Ready
   (c) Running (d) Blocked
   (e) Terminated.
4. **Program counter :** It gives the address of the next instruction to be executed for this process.
5. **Registers :** They include accumulator, general purpose registers, index registers, segment registers etc. Whenever a processor switches over from one process to another process, information about current status of the old process is saved in PCB so that the process may be restarted later on.

The process scheduling consists of the following sub-functions :

1. **Scheduling :** Select the process to be executed next on the CPU.
2. **Dispatching :** Set up execution of the selected process on the CPU.
3. **Context save :** Save the status of a running process when its execution is to be suspended.

Passing of CPU control from process1 to process2 and then back to process1 is shown in Fig. 8.2.2.



(S8.2)**Fig. 8.2.2 : Context switching**

Scanned with CamScanner

Syllabus Topic : Threads

8.3 Threads

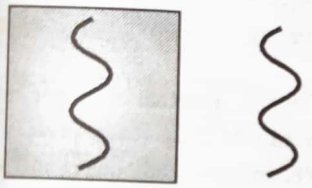
→ **(Dec. 2014, May 2015, Dec. 2015)**

Q. Explain different models of threads.
SPPU - Dec. 2014, May 2015, 6 Marks

Q. Explain threads in detail. **SPPU - Dec. 2015, 3 Marks**

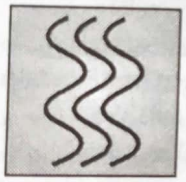
Threads are low cost alternative to processes for certain kind of concurrent applications. Each process has an address space and a single thread of control. In many cases, it is desirable to have multiple threads of control that share a single address space, but run in quasi-parallel, as if they were in fact separate processes (except for shared address space).

- Traditionally, there is single thread of execution per process (Fig. 8.3.1).



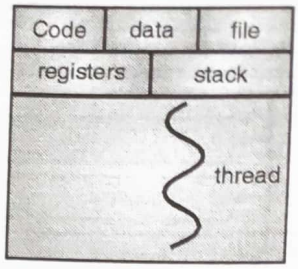
(S8.3) **Fig. 8.3.1 : One process one thread**

- In multithreading environment, O.S. supports multiple threads of execution within a single process (Fig. 8.3.2).

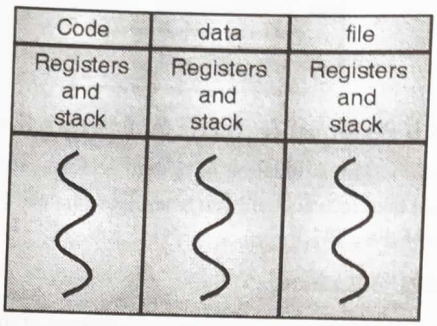


(S8.4) **Fig. 8.3.2 : One process multiple threads**

- A process is defined as the unit of resource allocation and a unit of protection. The followings are associated with a process :
  1. A virtual address space holding the process image.
  2. Protected access to processors, files and I/O resources.
- Within a process, there may be one or more threads, each with the following :
  1. A thread execution state (Running, ready, suspended, etc.)
  2. A process control block (PCB) for each thread for saving the thread context.
- Let us consider a web server application. A web server may have several clients accessing it. One solution is to have the server run as a single process that accepts requests. When the server receives a request it creates a separate process to service the request. This approach is quite slow as each process has its own code, data and files. It is more efficient to create a thread to service the request. Multiple threads created by a process share code, data and files. It is shown in Fig. 8.3.3.



(a) Single threaded process



(b) Multithreaded process

(S8.5) **Fig. 8.3.3 : Single threaded and multithreaded process**

8.3.1 Benefits of Threads

1. It takes less time to create a thread in an existing process than to create a new process. It improves responsiveness.
2. It takes less time to terminate thread than a process.
3. It takes less time to switch between two threads within the same process.
4. Threads share the memory and the resources of the process to which they belong. It allows an application to have several different threads of activity within the same address space.
5. Threads enhance efficiency in communication. They can interact with each other without the intervention of operating system.

Syllabus Topic : Scheduling

8.4 Process Scheduling

→ **(May 2016)**

Q. Explain process scheduling with neat diagram.
SPPU - May 2016, 6 Marks

Process scheduling is a fundamental operating system function. Scheduling is a set of policies and mechanisms supported by operating system that controls the order in which the work to be done is completed.

A scheduler is an operating system program that selects the next job to be admitted for execution. There are performance criteria that are frequently used by schedulers to maximize system performance. These are :

Scanned with CamScanner

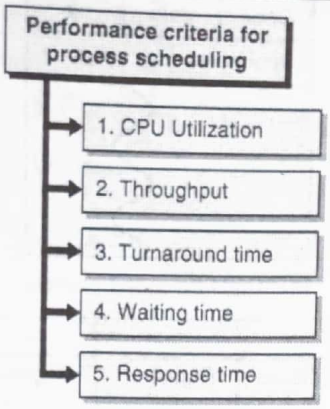


Fig. C8.1: Performance criteria for process scheduling

In addition to maximization of system performance, operating system should take into account **fairness**. That is, each process gets its fair share of the CPU.

➜ **1. CPU utilization**

The idea is to keep the CPU busy 100 percent of the time.

➜ **2. Throughput**

It refers to the number of processes executed per unit time.

➜ **3. Turnaround time**

It is defined as interval from the time of submission of a process to the time of its completion.

➜ **4. Waiting time**

The waiting time may be expressed as turnaround time, less the actual processing time

waiting time = turnaround time – processing time.

In multiprogramming operating system several jobs reside at a time in memory. CPU executes only one job at a time. The rest of jobs wait for the CPU.

➜ **5. Response time**

Response time is an important parameter in time sharing system. It is defined as interval from the time the last character of a command line of a program is entered to the last result appears on the terminal.

Throughput and CPU utilisation may be increased by executing a large number of processes, but then response time may suffer. Therefore, the scheduler must keep balance of all the different requirements and constraints.

8.4.1 Types of Schedulers

There are three types of schedulers :

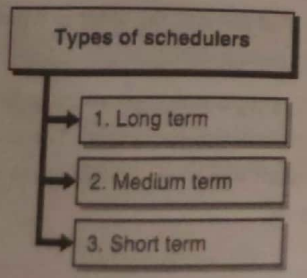
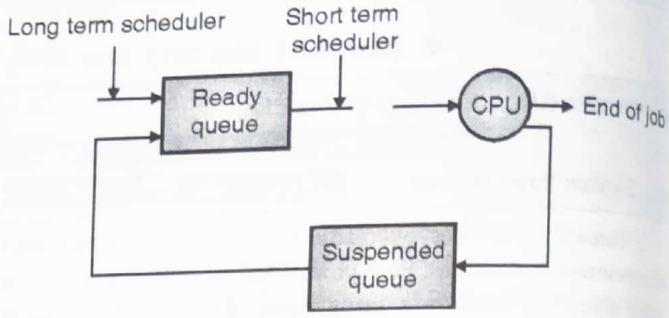


Fig. C8.2: Types of schedulers

➜ **1. Long term scheduler**

Long term scheduler is also called job scheduler. This determines which job shall be admitted for processing.



(S8.10) **Fig. 8.4.1 : Long term and short term scheduler**

There are always more processes than it can be accommodated by CPU in memory. These processes are kept waiting in large storage devices like disk for later processing. Whenever a process finishes, a long term scheduler is called. It selects a new process from among the waiting processes using SJF or FCFS strategy.

- Long term scheduler should maintain a proper mixture of CPU bound and I/O bound jobs in the ready queue.
- If all processes are I/O bound, the ready queue will always be empty.
- If all processes are CPU bound then system response will become poor.

➜ **2. Short term scheduler**

It allocates processes belonging to ready queue to CPU for immediate processing. Short term scheduler is called very often :

1. Called at the end of time slab.
2. Called in case of I/O request.

➜ **3. Medium term scheduler**

Most of the processes require I/O operation. In that case, it may become suspended for I/O operation after running a while. It is beneficial to remove these processes (suspended) from main memory to hard disk to create more memory for other processes. At some later time these processes can be reloaded into memory. The suspended processes are swapped out and swapped in by medium term scheduler.

Syllabus Topic : Types of Scheduling : Preemptive, Non Preemptive

8.5 Scheduling Methods

➜ **(May 2014, Dec. 2015)**

Q. Explain scheduling criteria ? Explain different types of scheduling algorithms in brief. **SPPU - May 2014, 8 Marks**

Q. What is CPU scheduling? Explain 2 different scheduling algorithms with examples. **SPPU - Dec. 2015, 6 Marks**

Scanned with CamScanner

CPU scheduling algorithms deals with the problem of deciding which of the processes in the ready queue to be allocated the CPU.

Scheduling algorithms can be classified into two categories :

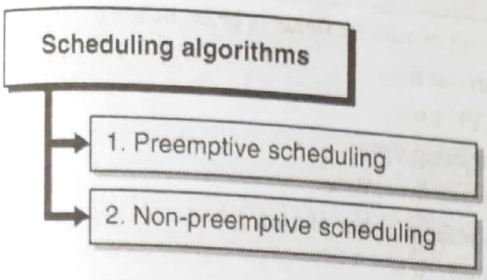


Fig. C8.3 : Scheduling algorithms

1. Preemptive scheduling

(May 2013)

Q. Explain preemptive concept with example. **SPPU - May 2013, 2 Marks**

In preemptive scheduling, the CPU can be taken away from the process. A process can be temporarily suspended due to :

1. Request for I/O.
2. Time slab is over.

2. Non-preemptive scheduling

(May 2013)

Q. Explain non preemptive concept with example. **SPPU - May 2013, 2 Marks**

In non-preemptive scheduling, a process runs to its completion. Once a process has been given the CPU, the CPU cannot be taken away from that process.

- Non-preemptive algorithms are simple and easy to implement.
- Non-preemptive algorithms are not suitable in multi-user environment.

Some of the common scheduling algorithms include :

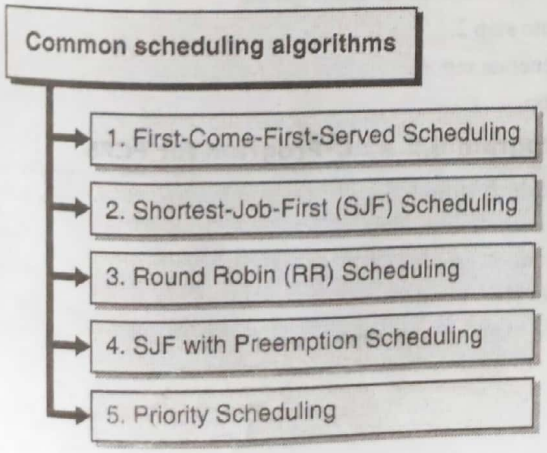


Fig. C 8.4 : Scheduling

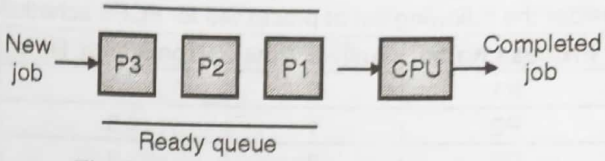
Syllabus Topic : Scheduling Algorithms - FCFS

8.5.1 First-Come-First-Served (FCFS) Scheduling

(May 2017)

Q. Explain following CPU scheduling techniques with example : FCFS. **SPPU - May 2017, 3 Marks**

It is a long term scheduling algorithm. It is one of the simplest scheduling algorithms, in which jobs are executed in the order of their arrival (Fig. 8.5.1).



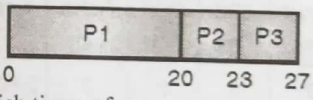
(S8.11) **Fig. 8.5.1 : First-Come-First-Served scheduling**

Its implementation involves maintaining a queue of arriving jobs. Whenever the CPU is free, a new job from the ready queue is assigned CPU. Once a process has the CPU, it runs to completion.

Consider the following example of three processes :

Process	Burst Time / Execution Time
P1	20
P2	3
P3	4

If the processes arrive in the order P1, P2 and P3 in quick succession and are served in FCFS order, we get the result shown in the following Gantt chart.



Start and finish times of processes are :

Process	Start time	CPU Time	Finish time
P1	0	20	20
P2	20	3	23
P3	23	4	27

The two important parameters :

1. Turnaround time.
2. Waiting time.

Which can be calculated as given below :

Turnaround time = Finish time – Arrival time

Waiting time = Turnaround time – CPU time.

A new table with turn around time and waiting time is shown in Table 8.5.1.

Table 8.5.1

Sr. No.	Process No.	CPU time	Start time	Finish time	Turnaround time	Waiting time
1.	P1	20	0	20	20	0
2.	P2	3	20	23	23	20
3.	P3	4	23	27	27	23
				Average →	70/3	43/3
					= 23.33	= 14.33

Scanned with CamScanner

Conclusions

1. FCFS algorithm is simple to implement.
2. FCFS algorithm results in poor performance. Average turnaround and waiting time are quite high.
3. FCFS meets the fairness criteria of scheduling.
4. FCFS algorithm can not be used in time sharing system, where it is important that each user gets a share of the CPU at regular interval. It is not desirable to allow one process to keep the CPU for a long time.

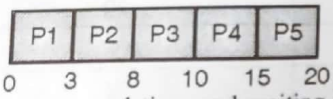
Example 8.5.1

Consider the following set of processes for FCFS scheduling

Process Name	Arrival Time	Processing Time
P1	0	3
P2	1	5
P3	3	2
P4	9	5
P5	12	5

Find waiting time and turnaround time of each process, also calculate the average turnaround time and waiting time.

Solution : The processing of the five processes is shown in the following Gantt chart :



A table with turnaround time and waiting time is shown in Table Ex. 8.5.1.

Table Ex. 8.5.1

Sr. No.	Process No.	Arrival Time	CPU Time	Start Time	Finish Time	Turnaround Time	Waiting Time
1.	P1	0	3	0	3	3 – 0 = 3	3 – 3 = 0
2.	P2	1	5	3	8	8 – 1 = 7	7 – 5 = 2
3.	P3	3	2	8	10	10 – 3 = 7	7 – 2 = 5
4.	P4	9	5	10	15	15 – 9 = 6	6 – 5 = 1
5.	P5	12	5	15	20	20 – 12 = 8	8 – 5 = 3
			Average →			31/5	11/5
						= 6.2	= 2.2

It may be noted that :

Turnaround time = Finish time – Arrival time

Waiting time = Turnaround time – CPU time

8.5.1(a) FCFS Algorithm

Databases used

- The algorithm requires three queues :
  1. Wait – Queue of new jobs.
  2. Ready – Queue of ready jobs.
  3. Finish – Queue of executed (Finished) jobs.
- A job enters the ready queue from the wait queue after Arrival time ≤ Current time

Each element of the queue is a structure as given below :

```
typedef struct node
   {
     int at, st, ft, wt, tat, sno ;
   } node ;
```

Meaning of various fields is given below ;

at – arrival time

st – CPU time

wt – waiting time

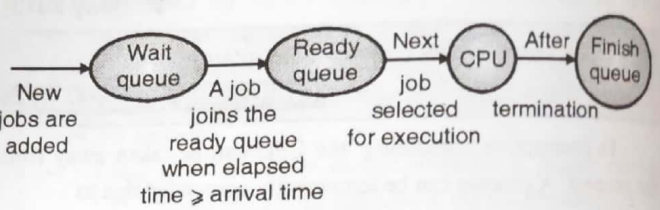
tat – turnaround time

sno – serial number of each job

ft – finish time.

The variable 'elapsed' gives the current time. Initial time is always taken as zero.

Flow of a job through various queues is shown in Fig. 8.5.2.



(S8.14)**Fig. 8.5.2**

Steps

1. Jobs to be processed are stored in the arrival queue.
   elapsed = 0
2. If both wait queue and the ready queue are empty then goto step 7.
3. For every job P in wait queue
   If P·at ≤ elapsed
   then
   P is deleted from wait queue and inserted in ready queue.
4. A new job P is selected from the ready queue for execution. Its finish-time, turnaround time and waiting time are calculated.
   elapsed = elapsed + P·st
5. Job P is inserted in finish-queue.
6. Goto step 2.
7. Generate report.
8. Stop.

Program 8.5.1 : C-Program for FCFS

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
typedef struct node
{   int at,st,ft,wt,tat,sno;

}node;

typedef struct Q
{   node data[30];
    int R,F;
```

Scanned with CamScanner

```
Q;

void init(Q *p)
{  p->R=p->F=-1;
}

void print(Q *q)
{  int i; node *p;
   printf("\n sno  Arrival  CPU Time  Waiting  Finish
Turn-around");
   printf("\n     Time          Time     Time     Time");
   printf("\n-----------------------------------------n");
   for(i=q->F;i<=q->R;i++)
   {
   p=q->data + i;
   printf("\n%d%2d%2d%2d%2d%2d      ",p->sno,p->
   at,p->st,p->wt,p->ft,p->tat);
   }
}

node front(Q *p)
{
   return(p->data[p->F]);
}

int empty(Q *p)
{  if(p->F==-1)
     return(1);
   return(0);
}

node Delete(Q *p)
{  node q;
   q=p->data[p->F];
   if(p->F==p->R)
     init(p);
   else
     p->F += 1;
   return(q);
}

void insert(Q *p,node newnode)
{
   if(empty(p))
     p->R=p->F=0;
   else
     p->R += 1;
     p->data[p->R]=newnode;
}

void main()
{  Q wait,ready,finish;
   node p,readynode,p1;
   int n,i,elapsed;
   clrscr();
   init(&wait);
   printf("\n enter no of processes: ");
   scanf("%d",&n);
   printf("\nEnter Arrival time, CPU time : \n");

   for(i=0;i<n;i++)
   {
     scanf("%d%d",&(p.at),&(p.st));
     p.ft=p.wt=0;
     p.sno=i+1;
     insert(&wait,p);
   }
            elapsed=0;
            init(&finish);
            init(&ready);
            while(!empty(&wait) || !empty(&ready))
            {
               while(!empty(&wait))
               {
                  p=front(&wait);
                  if(p.at<=elapsed)
                  {
                     p=Delete(&wait);
                     insert(&ready,p);
                  }
                  else
                     break;
               }
               if(empty(&ready))
                  elapsed++;
               else
               {
                  p=Delete(&ready);
                  p.ft=elapsed+p.st;
                  elapsed=p.ft;
                  p.tat=p.ft - p.at;
                  p.wt=p.tat - p.st;
                  insert(&finish,p);
               }
            }
            print(&finish);
}
```

Syllabus Topic : Scheduling Algorithms - SJF

8.5.2 Shortest-Job-First (SJF) Scheduling

(May 2017)

Q. Explain following CPU scheduling techniques with example : SJF. **SPPU - May 2017, 3 Marks**

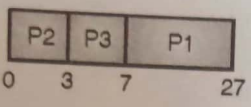
In Shortest-Job-First (SJF) scheduling, a job with shortest execution time is selected for execution. If two processes have the same CPU time, FCFS is used.

Scanned with CamScanner

Consider the following example of three processes :

Process	Burst time / Execution time
P1	20
P2	3
P3	4

Using SJF scheduling, we would schedule these processes according to the following Gantt chart :



- Shortest job P2 is selected for execution at time 0.
- Next shorter job P3 is selected for execution at time 3. The job P2 finished at time 3.
- Finally, the job P1 is selected for execution at time 7.

Start and finish times of processes are :

Table 8.5.2

Process	CPU Time	Start Time	Finish Time
P1	20	7	27
P2	3	0	3
P3	4	3	7

A new table with turnaround time and waiting time is shown in Table 8.5.3.

Table 8.5.3

Sr. No.	Process No.	CPU Time	Start Time	Finish Time	Turnaround Time	Waiting Time
1	P1	20	7	27	27	7
2	P2	3	0	3	3	0
3	P3	4	3	7	7	3
				Average →	37/3 = 12.33	10/3 = 3.33

- SJF algorithm is an optimal scheduling algorithm in terms of minimizing.
  1. The average waiting time.
  2. The average turnaround time.
- SJF algorithm does not guarantee fairness. If shorter jobs keep joining the ready then a relatively longer job may have to wait infinitely.
- The optimal performance of SJF scheduling is dependent upon future knowledge of the process/job behaviour. It is very difficult to estimate the exact CPU time requirement of a job/process.

Example 8.5.2

Consider the following set of processes for SJF scheduling :

Process Name	Arrival Time	Processing Time
P1	0	3
P2	1	5
P3	3	2
P4	9	5
P5	12	5

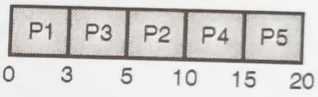
Find waiting time and turnaround time of each process.

Solution :

Table Ex. 8.5.2

Elapse Time	Process in the ready queue	Process selected for execution	Comment
0	P1	P1	P1 arrives at time 0 and needs 3 units of time
3	P3, P2	P3	Both P2 and P3 arrive by time 3. Shorter of the two, P3 is selected.
5	P2	P2	P3 and P4 do not arrive by time 5.
10	P4	P4	P4 arrives at time 9.
15	P5	P5	

Thus, processes will be executed in the following sequence. It is being shown through a Gantt chart :



A table with turnaround time and waiting time is shown in Table Ex. 8.5.2(a).

Table Ex. 8.5.2(a)

Sr. No.	Process No.	Arrival Time	CPU Time	Start Time	Finish Time	Turnaround Time	Waiting Time
1.	P1	0	3	0	3	3	0
2.	P2	1	5	5	10	9	4
3.	P3	3	2	3	5	2	0
4.	P4	9	5	10	15	6	1
5.	P5	12	5	15	20	8	3
					Average →	28/5 = 5.6	8/5 = 1.6

8.5.2(a) SJF Algorithm

1. Jobs to be processed are stored in the arrival queue.
2. Elapsed = 0.
3. If both arrival queue and the ready queue are empty then goto step 9.
4. for every job P in arrival queue.
   if P.at ≤ elapsed
   then
   P is deleted from the arrival queue and inserted in the ready queue.
5. Jobs in the ready queue are sorted with the smallest job at the front.
6. A new job P is selected from the ready queue for execution. Its finish-time, turnaround time and waiting time are calculated.
   elapsed = elapsed + P. st
7. Job P in inserted in finish-queue.

Scanned with CamScanner

8. goto step 3
9. Generate report
10. Stop.

Program 8.5.2 : C-Program for SJF :

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
typedef struct node
{   int at,st,ft,wt,tat,sno;

}node;

typedef struct Q
{   node data[30];
    int R,F;
}Q;

void init(Q *p)
{   p->R=p->F=-1;
}

void print(Q *q)
{   int i;  node *p;
    printf("\n sno  Arrival  CPU Time   Waiting  Finish
Turn-around");
    printf("\n     Time              Time    Time    Time");
    printf("\n-----------------------------------------------\n");
    for(i=q->F;i<=q->R;i++)
    {
    p=q->data + i;
    printf("\n%d%2d%2d%2d%2d%2d        ",p->sno,p->
    at,p->st,p->wt,p->ft,p->tat);
    }
}



node front(Q *p)
{
    return(p->data[p->F]);
}

int empty(Q *p)
{   if(p->F==-1)
      return(1);
    return(0);
}

node Delete(Q *p)
{   node q;
    q=p->data[p->F];
    if(p->F==p->R)
      init(p);
    else
      p->F += 1;
    return(q);
}

void insert(Q *p,node newnode)
{
    if(empty(p))
      p->R=p->F=0;
    else
      p->R += 1;
      p->data[p->R]=newnode;
}

void sort(Q *p)
{   node *a,temp;
    int i,j,n;
    a=p->data;
    n=p->R - p->F +1;
    for(i=1;i<n;i++)
      for(j=0;j<n-i;j++)
        if(a[j+p->F].st > a[j+p->F+1].st)
        {
            temp=a[j+p->F];
            a[j+p->F]=a[j+p->F+1];
            a[j+p->F +1 ]=temp;
        }
}

void main()
{   Q wait,ready,finish;
    node p,readynode,p1;
    int n,i,elapsed;
    clrscr();
    init(&wait);
    printf("\n enter no of processes: ");
    scanf("%d",&n);
    printf("\nEnter Arrival time, CPU time : \n");

    for(i=0;i<n;i++)
    {
      scanf("%d%d",&(p.at),&(p.st));
      p.ft=p.wt=0;
      p.sno=i+1;
      insert(&wait,p);
    }
            elapsed=0;
            init(&finish);
            init(&ready);
```

Scanned with CamScanner

```
while(!empty(&wait) || !empty(&ready))
{
    while(!empty(&wait))
    {
        p=front(&wait);
        if(p.at<=elapsed)
        {
            p=Delete(&wait);
            insert(&ready,p);
        }
        else
            break;
    }
    sort(&ready);
    if(empty(&ready))
        elapsed++;
    else
    {
        p=Delete(&ready);
        p.ft=elapsed+p.st;
        elapsed=p.ft;
        p.tat=p.ft - p.at;
        p.wt=p.tat - p.st;
        insert(&finish,p);
    }
}
print(&finish);
}
```

Syllabus Topic : Scheduling Algorithms - RR

8.5.3 Round Robin (RR) Scheduling

→ (May 2013)

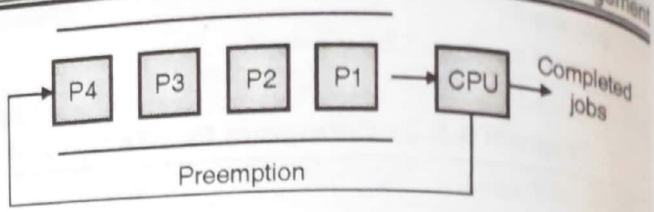
Q. Write short notes on : Round Robin Scheduling.

SPPU - May 2013, 3 Marks

The round robin scheduling algorithm is primarily used in a time-sharing and a multi-user system. The basic requirement is such systems is to provide reasonably good response time and in general to share the system fairly among all system users.

In round robin scheduling, the CPU time is divided in slices (quantum). Each process is allocated one time-slice, while it is running.

- A process is given one time slice at a time for execution.
- After the end of time slice, the process goes to the end of ready queue and a process from the front of ready queue is selected for execution (as shown in Fig. 8.5.3).



(S8.17) **Fig. 8.5.3 : Round robin scheduling**

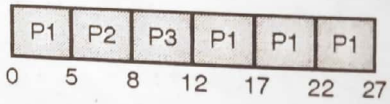
- A time slice is usually of 10 to 100 milliseconds.
- Setting the quantum too short causes too many process switches and lowers the CPU efficiency. Setting the quantum too long may cause poor response to short interactive requests.
- Round robin scheduling utilises the system resources in an equitable manner. Small process may be executed in a single time-slice giving good response time whereas long processes may require several time slices and thus be forced to pass through ready queue a few times before completion.

Consider the following example of three processes :

Process	Execution time
P1	20
P2	3
P3	4

Time slice (quantum) = 5 units of time

Using round robin scheduling, we would schedule these processes according to the following Gantt chart.



- P1 gets first 5 units of time, it is pre-empted and the CPU is given to next job.
- P2 gets 3 units of time as it does not need 5 units of time. P2 finishes at time = 8.
- P3 gets 4 units of time as it does not need 5 units of time. P3 finishes at time = 12.
- The CPU is returned to P1 for additional three time slices.

Finish time of various processes are :

Process	CPU time	Finish time
P1	20	27
P2	3	8
P3	4	12

A new table with turnaround time and waiting time is shown in Table 8.5.4.

Table 8.5.4

S. No.	Process No.	CPU time	Finish time	Turnaround time	Waiting time
1	P1	20	27	27	7
2	P2	3	8	8	5
3	P2	4	12	12	8
			Average →	47/3 = 15.66	20/3 = 6.66

Scanned with CamScanner

Example 8.5.3

Consider the following set of processes for round robin scheduling :

Process Name	Arrival Time	Processing Time
P1	0	3
P2	1	5
P3	3	2
P4	9	5
P5	12	5

Find waiting time and turnaround time of each process. Assume time slice = 2 units.

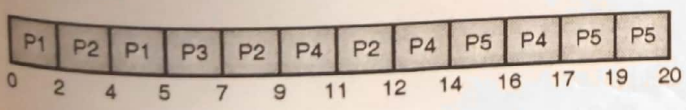
Solution :

Step 1 : Finding completion time of each process.

Table Ex. 8.5.3

Time	Ready Queue	Next job to be executed
0	P1	P1 for 2 time units
2	P2 → P1	P2 for 2 time units
4	P1 → P3 → P2	P1 for 1 time unit P1 finishes
5	P3→ P2	P3 for 2 time units, P3 finishes.
7	P2	P2 for 2 time units
9	P4 → P2	P4 for 2 time units
11	P2 → P4	P2 for 1 time unit. P2 finishes
12	P4 → P5	P4 for 2 time units
14	P5 → P4	P5 for 2 time units
16	P4 → P5	P4 for 1 time unit. P4 finishes
17	P5	P5 for 2 time units
19	P5	P5 for 1 time unit.

Gantt chart



Step 2 : A table with turnaround time and waiting time is shown in Table Ex. 8.5.3(a) :

Table Ex. 8.5.3(a)

Sr. No.	Process No.	Arrival time	CPU time	Finish time	Turnaround time	Waiting time
1	P1	0	3	5	5	2
2	P2	1	5	12	11	6
3	P3	3	2	7	4	2
4	P4	9	5	17	8	3
5	P5	12	5	20	8	3

8.5.3(a) Round Robin Algorithm

Databases used :

The algorithm requires three queues :

1. Wait – Queue of new(arriving) jobs.
2. Ready – Queue of ready jobs.
3. Finish – Queue of executed (finished) jobs.

– A job enters the ready queue from the wait-queue after

Arrival time ≤ current time

Each element of the queue is a structure as given below :

```
typedef struct node
{
    int at, st, ft, wt, et, tat, sno;
} node;
```

Meaning of various fields is given below :

at – arrival time

st – CPU time

ft – finish time

wt – waiting time

et – CPU time already given to the process

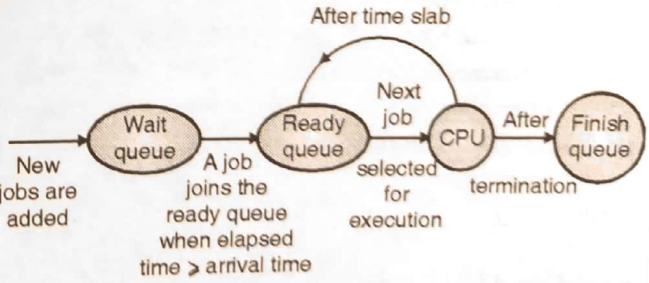
tat – Turnaround time

sno – serial number of each job.

The variable 'elapsed' gives the current time.

Initial time is always taken as zero.

The variable 'slab' gives the quantum of time. Flow of a process through various queues is shown in Fig. 8.5.4.



(S8.21)**Fig. 8.5.4**

Steps

1. Jobs to be processed are stored in the wait (arrival) queue.
   elapsed = 0
2. If both wait queue and the ready queue are empty then goto step 8.
3. For every job P in wait (arrival) queue
   if P. at ≤ elapsed
   then
   P is deleted from wait queue and inserted in ready queue.
4. If ready queue is empty then
   a. elapsed = elapsed + slab

Scanned with CamScanner

b. goto step 2

5. A process P is selected from the ready queue for execution.
   a. if P.et + slab ≥ P.st then

```
{
  elapsed = elapsed + P.st – P.et
  P.et = P.st
}
```

   b. If P.et + slab < P.st then

```
{
  1. P.et = P.et + slab
  2. elapsed = elapsed + slab
}
```

6. For every job P1 in wait (arrival) queue
   if P1.at ≤ elapsed
   then
   P1 is deleted from wait queue and inserted in ready queue
7. If the process P has run to its completion (i.e. P.et = P.st) then
   a. P.ft = elapsed
   b. P.tat = P.ft – P.at
   c. P.wt = P.tat – P.st
   d. insert P in finish queue
   e. goto step 4.
8. Generate report
9. Stop.

➢ **Program 8.5.3 : C-program for Round Robin Scheduling**

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
typedef struct node
{   int at,st,ft,wt,et,tat,sno;
    int priority;
}node;

typedef struct Q
{   node data[30];
    int R,F;
}Q;

void init(Q *p)
{   p->R=p->F=-1;
}

void print(Q *q)
{   int i; node *p;
    printf("\n\nsno arrival service waiting finish turn-
    around\n");
    for(i=q->F;i<=q->R;i++)
    {
      p=q->data + i;
      printf("\n%d %2d %2d %2d %2d
      %2d ",p->sno,p->at, p->st,p->wt,p->ft,p->tat);
    }
}

node front(Q *p)
{
    return(p->data[p->F]);
}

int empty(Q *p)
{   if(p->F==-1)
      return(1);
    return(0);
}

node Delete(Q *p)
{   node q;
    q=p->data[p->F];
    if(p->F==p->R)
      init(p);
    else
      p->F += 1;
    return(q);
}

void insert(Q *p,node newnode)
{   node *q;
    if(empty(p))
      p->R=p->F=0;
    else
      p->R += 1;
      p->data[p->R]=newnode;
}


void main()
{   Q wait,ready,finish;
    node p,readynode,p1;
    int n,i,elapsed,slab;
    int start,end;
    clrscr();
    printf("\n enter no of processes: ");
    scanf("%d",&n);
    printf("\nEnter arrival time, service time: \n");
    init(&wait);
    for(i=0;i<n;i++)
    {
      scanf("%d%d",&(p.at),&(p.st));
      p.et=p.ft=p.wt=0;
```

Scanned with CamScanner

```
    p.sno=i+1;
    insert(&wait,p);
  }

  printf("\nEnter Time Slab : ");
  scanf("%d",&slab);
  elapsed=0;
  init(&finish);
  init(&ready);

  while(!empty(&wait)) //processes coming at time=0 are
ved to ready queue
    {
    p=front(&wait);
    if(p.at<=elapsed)
     {
        p=Delete(&wait);
        insert(&ready,p);
     }
    else
        break;
    }

printf("\nGantt Chart :\n");
while(!empty(&wait) || !empty(&ready) ) //main loop
cheduling
  {
  if(empty(&ready))
    {
    elapsed++;
    while(!empty(&wait))    //processes coming at
sed time are moved to ready queue
     {
        p=front(&wait);
        if(p.at<=elapsed)
         {
            p=Delete(&wait);
            insert(&ready,p);
         }
        else
            break;
     }

    }
  else
    {  start=elapsed;//for Gantt chart
    p=Delete(&ready);
    if(slab > p.st - p.et)
     {
        elapsed=elapsed+p.st - p.et;
        p.et= p.st;
         }
        else
         {
            p.et=p.et+slab;
            elapsed=elapsed+slab;
         }

        while(!empty(&wait))
                        //add new jobs to ready queue
            {
                p1=front(&wait);
                if(p1.at<=elapsed)
                 {
                   p1=Delete(&wait);
                   insert(&ready,p1);
                 }
                else
                   break;
            }

        if(p.et==p.st)
         {
            p.ft=elapsed;
            p.tat=p.ft - p.at;
            p.wt=p.tat - p.st;
            insert(&finish,p);
         }
        else
         {
            insert(&ready,p);
         }
        end=elapsed;
        printf("(%d-P%d-%d)",start,p.sno,end);
        }
  }
  print(&finish);
  getch();

}
```

➜ 8.5.4 SJF with Preemption Scheduling

It is a preemptive version of SJF. In this scheduling, the scheduler always selects a process for execution with the shortest expected remaining processing time.

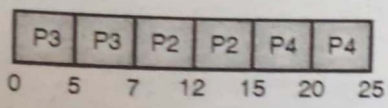
- A process is preempted after the time slab (quantum).
- The next process to be scheduled for execution is based on shortest expected remaining processing time.

Consider the following example of three processes :

Process	Execution
P1	10
P2	8
P3	7

Scanned with CamScanner

Time slice (quantum) = 5 units of time using preemptive SJF, we would schedule these processes according to the following Gantt chart.



- P3 being the shortest process is selected for execution. It requires 2 more units of time.
- P3 is again selected for execution. It runs to its completion.
- After P3, the next shorter job P2 is selected for execution. It requires a full time slab and a part of the next slab. It runs to its completion.
- Finally P1 is selected for execution.

Finish time of various processes are :

Process	CPU Time	Finish Time
P1	10	25
P2	8	15
P3	7	7

A new table with turnaround time and waiting time is shown in Table 8.5.5.

Table 8.5.5

Sr. No	Process No.	CPU Time	Finish Time	Turnaround Time	Waiting Time
1.	P1	10	25	25	15
2.	P2	8	15	15	7
3.	P3	7	7	7	0

Example 8.5.4

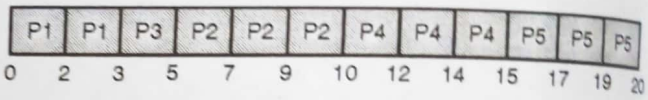
Consider the following set of processes for preemptive SJF scheduling (Time slab 2) :

Process Name	Arrival Time	Processing Time
P1	0	3
P2	1	5
P3	3	2
P4	9	5
P5	12	5

Solution : Find waiting time and turnaround time of each process:

Time	Real Queue (smallest process at the front)	Next job to be executed
0	P1	P1 for 2 units. P1 needs additional 1 unit of time.
2	P1 P2	P1 for 1 unit. P1 completes
3	P3 P2	P3 for 2 units. P3 completes
5	P2	P2 for 2 unit
7	P2	P2 for 2 units
9	P2 P4	P2 for 1 unit P2 completes
10	P4	P4 for 2 units
12	P4 P5	P4 for 2 units
14	P4 P5	P4 for 1 unit P4 completes.
15	P5	P5 for 2 units
17	P5	P5 for 2 units
19	P5	P5 for 1 unit P5 completes.

Gantt Chart



A table with turnaround time and waiting time is shown in Table Ex. 8.5.4.

Table Ex. 8.5.4

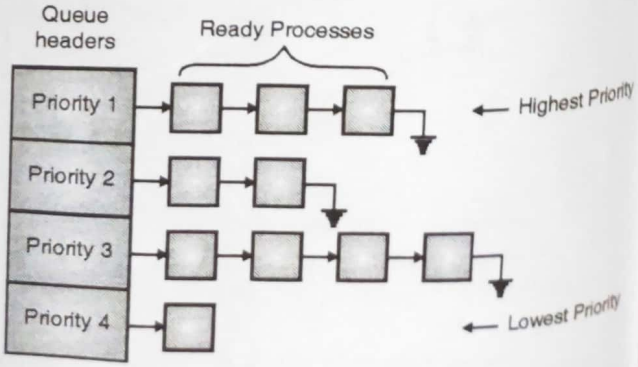
Sr. No	Process No.	Arrival Time	CPU Time	Finish Time	Turnaround Time	Waiting Time
1.	P1	0	3	3	3	0
2.	P2	1	5	10	9	4
3.	P3	3	2	5	2	0
4.	P4	9	5	15	6	1
5.	P5	12	5	20	8	3

8.5.5 Priority Scheduling

Priority scheduling is preemptive scheduling method. Each process is assigned a priority. Among the runnable processes, one with the highest priority is allowed to run first.

To prevent high-priority processes from waiting indefinitely, the scheduler may decrease the priority of the currently running process after every time slab. Fig. 8.5.5 shows a figure with four priority classes.

It is often necessary to group processes into priority classes. Round robin scheduling is used within each priority class.



(S8.24) **Fig. 8.5.5 : Scheduling with four priority classes**

Scanned with CamScanner

Example 8.5.5

Consider the following set of jobs :

Job No.	Arrival time	Run time	Job No.
1	10	2	1
2	10	2	2
3	11	1	3
4	13	3	4
5	14	1	5

Determine the finish and turnaround time for the above jobs using following methods.

(i) First come first serve

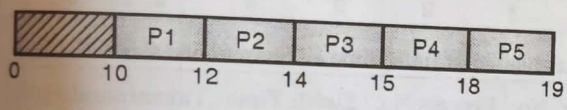
(ii) Shortest job first

(iii) Round Robin (Assume time quantum = 0.5)

Solution :

(i) First come first serve scheduling

Gantt chart for scheduling is shown below :

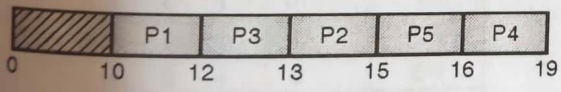


Calculation of turnaround time and finish time :

Sr. No.	Arrival Time	CPU Time	Start Time	Finish Time	Turnaround Time
1	10	2	10	12	2
2	10	2	12	14	4
3	11	1	14	15	4
4	13	3	15	18	5
5	14	1	18	19	5

(ii) Shortest job first

Gantt chart for scheduling is shown below :



Calculation of turnaround time and finish time :

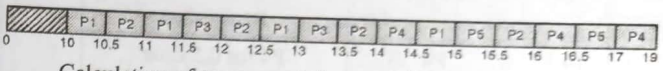
Sr. No.	Process No.	Arrival Time	CPU Time	Start Time	Finish Time	Turnaround Time
1	P1	10	2	10	12	2
2	P3	11	1	12	13	2
3	P2	10	2	13	15	5
4	P5	14	1	15	16	2
5	P4	13	3	16	19	6

(iii) Round Robin (Slab = 0.5)

Finding completion time of each process :

Time	Ready Queue	Next job to be executed
10	P1 P2	P1 for 0.5 time
10.5	P2 P1	P2 for 0.5 time
11	P1 P3 P2	P1 for 0.5 time
11.5	P3 P2 P1	P3 for 0.5 time
12	P2 P1 P3	P2 for 0.5 time
12.5	P1 P3 P2	P1 for 0.5 time
13	P3 P2 P4 P1	P3 for 0.5 time
13.5	P2 P4 P1	P2 for 0.5 time
14	P4 P1 P5 P2	P4 for 0.5 time
14.5	P1 P5 P2 P4	P1 for 0.5 time
15	P5 P2 P4	P5 for 0.5 time
15.5	P2 P4 P5	P2 for 0.5 time
16	P4 P5	P4 for 0.5 time
16.5	P5 P4	P5 for 0.5 time
17	P4	P4 for 2 time units

Gantt chart



Calculation of turnaround time and finish time :

Sr. No.	Process No.	Arrival Time	CPU Time	Finish Time	Turnaround Time
1	P3	11	1	13.5	2.5
2	P1	10	2	15	5
3	P2	10	2	16	6
4	P5	14	1	17	3
5	P4	13	3	19	6

Example 8.5.6

Consider the following set of jobs :

Job No.	Arrival time	Run time
1	10	2
2	10	1
3	11	1
4	12	1

Determine the turnaround time for the above jobs using following methods.

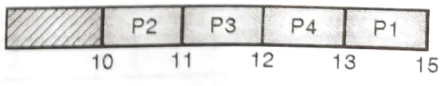
(i) Shortest job first

(ii) Round Robin (Assume time quantum = 0.5)

Solution :

(i) Shortest job first

Gantt chart for scheduling is shown below :



Calculation of turnaround time :

Sr. No.	Process No.	Arrival Time	Finish Time	Turnaround Time
1	P2	10	11	11 – 10 = 1
2	P3	11	12	12 – 11 = 1
3	P4	12	13	13 – 12 = 1
4	P1	10	15	15 – 10 = 5

Scanned with CamScanner

(iii) Round Robin (Time slab = 0.5)

Finding completion time of each process :

Time	Ready Queue	Next job to be executed
10	P1 P2	P1 for 0.5 time
10.5	P2 P1	P2 for 0.5 time
11	P1 P3 P2	P1 for 0.5 time
11.5	P3 P2 P1	P3 for 0.5 time
12	P2 P1 P4 P3	P2 for 0.5 time P2 finish
12.5	P1 P4 P3	P1 for 0.5 time
13	P4 P3 P1	P4 for 0.5 time
13.5	P3 P1 P4	P3 for 0.5 time P3 finishes
14	P1 P4	P1 for 0.5 time P1 finishes
14.5	P4	P4 for 0.5 time P4 finishes

Calculation of turnaround time

Sr. No.	Job No.	Arrival Time	Finish Time	Turnaround Time
1	P2	10	12.5	2.5
2	P3	11	14	3
3	P1	10	14.5	4.5
4	P4	12	15	3

Example 8.5.7

Consider the following set of jobs :

Job	Run time
A	2
B	1
C	1
D	2
E	1

Calculate finish time and turnaround time of these jobs using the following methods

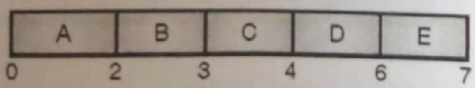
(i) First Come First Serve
(ii) Shortest Job First
(iii) Round Robin

Assume all jobs have an arrival time of 10.00.

Solution :

(i) First Come First Serve

Gantt chart

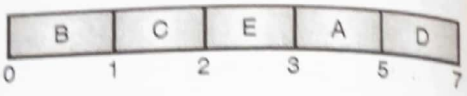


Calculation of finish time and turnaround time :

Sr. No.	Process No.	Finish Time	Turnaround Time
1	A	2	2
2	B	3	3
3	C	4	4
4	D	6	6
5	E	7	7

(ii) Shortest job first

Gantt chart

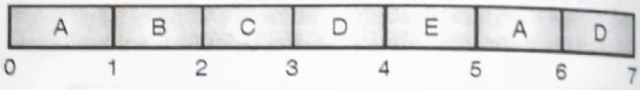


Calculation of finish time and turnaround time :

Sr. No.	Process No.	Finish Time	Turnaround Time
1	B	1	1
2	C	2	2
3	E	3	3
4	A	5	5
5	D	7	7

(iii) Round Robin (slab = 1)

Gantt chart



Calculation of finish time and turnaround time

Sr. No.	Process No.	Finish Time	Turnaround Time
1	B	2	2
2	C	3	3
3	E	5	5
4	A	6	6
5	D	7	7

Example 8.5.8

Consider following set of processes :

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

Assume arrival time for all processes is at time 0 in order P1, P2, P3, P4, P5.

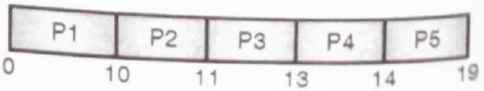
(i) Draw Gantt chart to show execution of these processes using FCFS, SJF and non-preemptive priority. (Assume small priority number implies higher priority).

(ii) Calculate turn-around and waiting time for each process using FCFS and SJF.

Solution :

(1) FCFS scheduling

Gantt chart



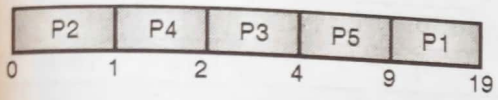
Calculation of turnaround time and waiting time

Scanned with CamScanner

Sr. No.	Process No.	Burst Time	Finish time	Turnaround time	Waiting Time
1	P1	10	10	10	0
2	P2	1	11	11	10
3	P3	2	13	13	11
4	P4	1	14	14	13
5	P5	5	19	19	14

(ii) SJF scheduling

Gantt chart

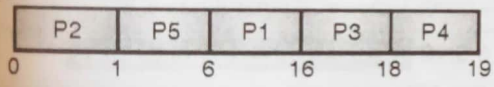


Calculation of turnaround time and waiting time :

Sr. No.	Process No.	Burst Time	Finish time	Turnaround time	Waiting Time
1	P2	1	1	1	0
2	P4	1	2	2	1
3	P3	2	4	4	2
4	P5	5	9	9	4
5	P1	10	19	19	9

(iii) Priority Scheduling (non-preemptive)

Gantt chart



Calculation of turnaround time and waiting time :

Sr. No.	Process No.	Burst Time	Finish time	Turnaround time	Waiting Time
1	P2	1	1	1	0
2	P5	5	6	6	1
3	P1	10	16	16	6
4	P3	2	18	18	16
5	P4	1	19	19	18

Example 8.5.9

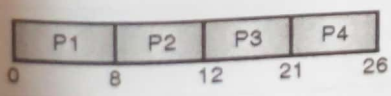
For the following set of processes calculate the average waiting time and turnaround time for FCFS, SJF and RR scheduling. Also draw Gantt chart.

Process No.	Arrival time	Burst time
1	0	8
2	1	4
3	2	9
4	3	5

Solution :

(i) FCFS scheduling

Gantt chart

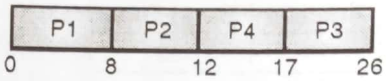


Calculation of turnaround time and waiting time

Sr. No.	Process No.	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
1	P1	0	8	8	8	0
2	P2	1	4	12	11	7
3	P3	2	9	21	19	10
4	P4	3	5	26	23	18

(ii) SJF scheduling

Gantt chart

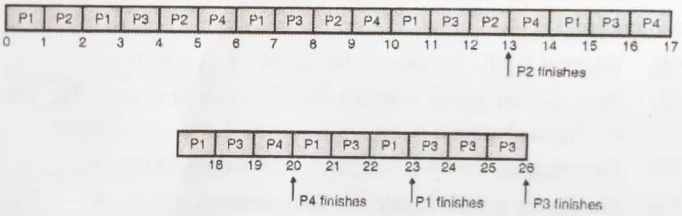


Calculation of turnaround time and waiting time :

Sr. No.	Process No.	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
1	P1	0	8	8	8	0
2	P2	1	4	12	11	7
3	P4	3	5	17	14	9
4	P3	2	9	26	24	15

(iii) RR scheduling (Slab = 1)

Gantt chart



Calculation of turnaround time and waiting time :

Sr. No.	Process No.	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
1	P2	1	4	13	12	8
2	P4	3	5	20	17	12
3	P1	0	8	23	23	15
4	P3	2	9	26	24	15

Example 8.5.10

Comment on the statement :

The weighted turnarounds in FCFS scheduling are arbitrarily distributed while those in SJF scheduling increase monotonically.

Solution : FCFS scheduling algorithm satisfies the following relationship :

$$\frac{T_r}{T_s} = \frac{1}{1-\rho}$$

Where T_r = Turnaround time

T_s = Average service time

ρ = Processor utilization

Scanned with CamScanner

- The performance of FCFS algorithm is found to be very unfavourable while considering weighted turnarounds. It will be dependent on the arrival rate and service requirement of incoming processes.
- In SJF scheduling, shorter processes have priority over longer processes. These shorter processes will keep the weighted turnarounds under check. Therefore, it will increase monotonically.

Example 8.5.11 **SPPU - May 2015, 6 Marks**

Consider the following processes where Arrival and Burst time are as "shown below :

Process	Burst Time
P_1	05
P_2	04
P_3	07
P_4	06

Calculate the Average Waiting Time and Average Turn-around Time if the processes are scheduled using SJF.

Solution : SJF Scheduling

Gantt chart

P_2	P_1	P_4	P_3

0 4 9 15 22

Calculation of the average waiting time and average turn-around time.

(1) Shortest job P2 is selected for execution at time 0.

(2) Next shorter job is selected for execution at time 4. The job P2 finished at time 4.

(3) Next shorter job P4 is selected for execution at time 9.

(4) Finally the job P3 is selected for execution at time 15.

Start and finish times of processes are

Table Ex. 8.5.11

Processes	CPU Time	Start Time	Finish Time
P_1	05	4	9
P_2	04	0	4
P_3	07	15	22
P_4	06	9	15

A new table with turnaround time and waiting time is shown in Table Ex. 8.5.11(a).

Table Ex. 8.5.11(a)

Sr. No.	Processes	CPU Time	Start Time	Finish Time	Turnaround Time	Waiting Time
1	P_1	05	4	9	9	4
2	P_2	04	0	4	4	0
3	P_3	07	15	22	22	15
4	P_4	06	9	15	15	9
	Average ⟶				50/4 = 12.5	28/4 = 7

Turnaround time = Finish time – Arrival time

Waiting time = Turnaround – CPU time

Example 8.5.12 **SPPU - May 2015, 6 Marks**

Consider the following processes where Arrival and Burst time (in seconds) are as shown below

Process	Burst Time	Arrival Time
P1	06	1
P2	04	4
P3	03	2
P4	06	5

Calculate the Average Waiting Time and Average turn-around Time if the processes are scheduled using FCFS.

Solution :

P1	P2	P3	P4

1 7 1114 20

Pr. No.	Arrival time	CPU time	Start time	Finish time	Turnaround time	Waiting time
P1	1	6	1	7	6	0
P2	4	4	7	11	7	3
P3	2	3	11	14	12	9
P4	5	6	14	20	15	9
					Avg = $\frac{40}{4} = 10\frac{21}{4} = 5.25$	

Turn around time = Finish – Arrival

Waiting Time = Turnaround – CPU

Example 8.5.13 **SPPU - Dec. 2016, 6 Marks**

Consider the following processes where Arrival and Burst time (in seconds) are as shown below

Process	Burst Time	Arrival Time
P1	06	0
P2	04	1
P3	07	3
P4	02	5

Calculate the Average Waiting Time and Average turn-around Time if the processes are scheduled using FCFS.

Solution :

Pr. No.	Arrival time	CPU time	Start time	Finish time	Turnaround time	Waiting time
P1	0	6	0	6	6	0
P2	1	4	6	10	9	5
P3	2	7	10	17	15	8
P4	3	2	17	19	16	14

Average Turnaround time = $\frac{6+9+15+16}{4} = \frac{46}{4}$

Average Waiting Time = $\frac{0+5+8+14}{4} = \frac{27}{4}$

Scanned with CamScanner

Exam Pack (University Questions)

Syllabus Topic : Process Concept

Explain process in detail.
(Refer section 8.1) (3 Marks) **(Dec. 2015)**

Syllabus Topic : Process States

Draw and explain process state transitions.
(Refer section 8.2.1) (6 Marks) **(May 2013)**

Explain various states of a process with diagram.
(Refer section 8.2.1) (6 Marks)
(Dec. 2014, May 2015, May 2016, Dec. 2016)

What are the various states of processes and how it is managed by operating system ?
(Refer section 8.2.1) (6 Marks) **(May 2017)**

Write short notes on : Process Control Block.
(Refer section 8.2.2) (3 Marks) **(May 2013)**

Draw and explain process control block.
(Refer section 8.2.2) (6 Marks) **(Dec. 2016)**

Syllabus Topic : Threads

Explain different models of threads.
(Refer section 8.3) (6 Marks) **(Dec. 2014, May 2015)**

Explain threads in detail.
(Refer section 8.3) (3 Marks) **(Dec. 2015)**

Syllabus Topic : Scheduling

Explain process scheduling with neat diagram.
(Refer section 8.4) (6 Marks) **(May 2016)**

Syllabus Topic : Types of Scheduling : Preemptive, Non Preemptive

Q. Explain scheduling criteria ? Explain different types of scheduling algorithms in brief.
(Refer section 8.5) (8 Marks) **(May 2014)**

Q. What is CPU scheduling? Explain 2 different scheduling algorithms with examples.
(Refer section 8.5) (6 Marks) **(Dec. 2015)**

Q. Explain preemptive concept with example.
(Refer section 8.5(1)) (2 Marks) **(May 2013)**

Q. Explain non preemptive concept with example.
(Refer section 8.5(2)) (2 Marks) **(May 2013)**

Syllabus Topic : Scheduling Algorithms - FCFS

Q. Explain following CPU scheduling techniques with example : FCFS. *(Refer section 8.5.1)* (3 Marks)
(May 2017)

Syllabus Topic : Scheduling Algorithms - SJF

Q. Explain following CPU scheduling techniques with example : SJF. *(Refer section 8.5.2)* (3 Marks)
(May 2017)

Syllabus Topic : Scheduling Algorithms - RR

Q. Write short notes on : Round Robin Scheduling.
(Refer section 8.5.3) (3 Marks) **(May 2013)**

Example 8.5.11 (6 Marks) **(May 2015)**

Example 8.5.12 (6 Marks) **(May 2015)**

Example 8.5.13 (6 Marks) **(Dec. 2016)**

❑❑❑

Scanned with CamScanner