Unit 1

What is System Programming?

System Programming consists of a variety of programs that support the operations of a computer. This software makes it possible for the user to focus on an application or the other problem to be solved. System programs (e.g. compiler, loader macro processors, and operating systems) were developed to make computer better adapted d to the need of their users.

Component of System Programming

Components of system programming are

1. Assemble
2. Loader
3. Compiler
4. Macros
5. Formal System

Assembler: -

- An assembler is a type of computer program that interprets software programs written in assembly language into machine language, code and instructions that can be executed by a computer.
- An assembler enables software and application developers to access, operate and manage a computer's hardware architecture and components.
- An assembler is sometimes referred to as the compiler of assembly language. It also provides the services of an interpreter.

Loader:-

- A *loader* is a program used by an operating system to load programs from a secondary to main memory so as to be executed.

Compiler:-

A *compiler* is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another

computer language (the target language), with the latter often having a binary form known as object code.

Macro:-

- A macro is a single line abbreviation for group of statement.
- A macro processor is a program that substitutes and specialized macro definitions for macro calls.

Formal System:-

- A *formal system* consists of a **language** over some **alphabet** of **symbols** together with (**axioms** and) **inference rules** that distinguish some of the strings in the language as **theorems**.
- A formal system has the following components:
    - A **finite alphabet** of symbols.
    - A **syntax** that defines which strings of symbol are in the language of our formal system.
    - A **decidable** set of **axioms** and a finite set of **rules** from which the set of **theorems** of the system is generated. The rules must take a finite number of steps to apply.

----*-----**

General Machine Structure

All the conventional modern computers are based upon the concept of stored program computer, the model that was proposed by John von Neumann.

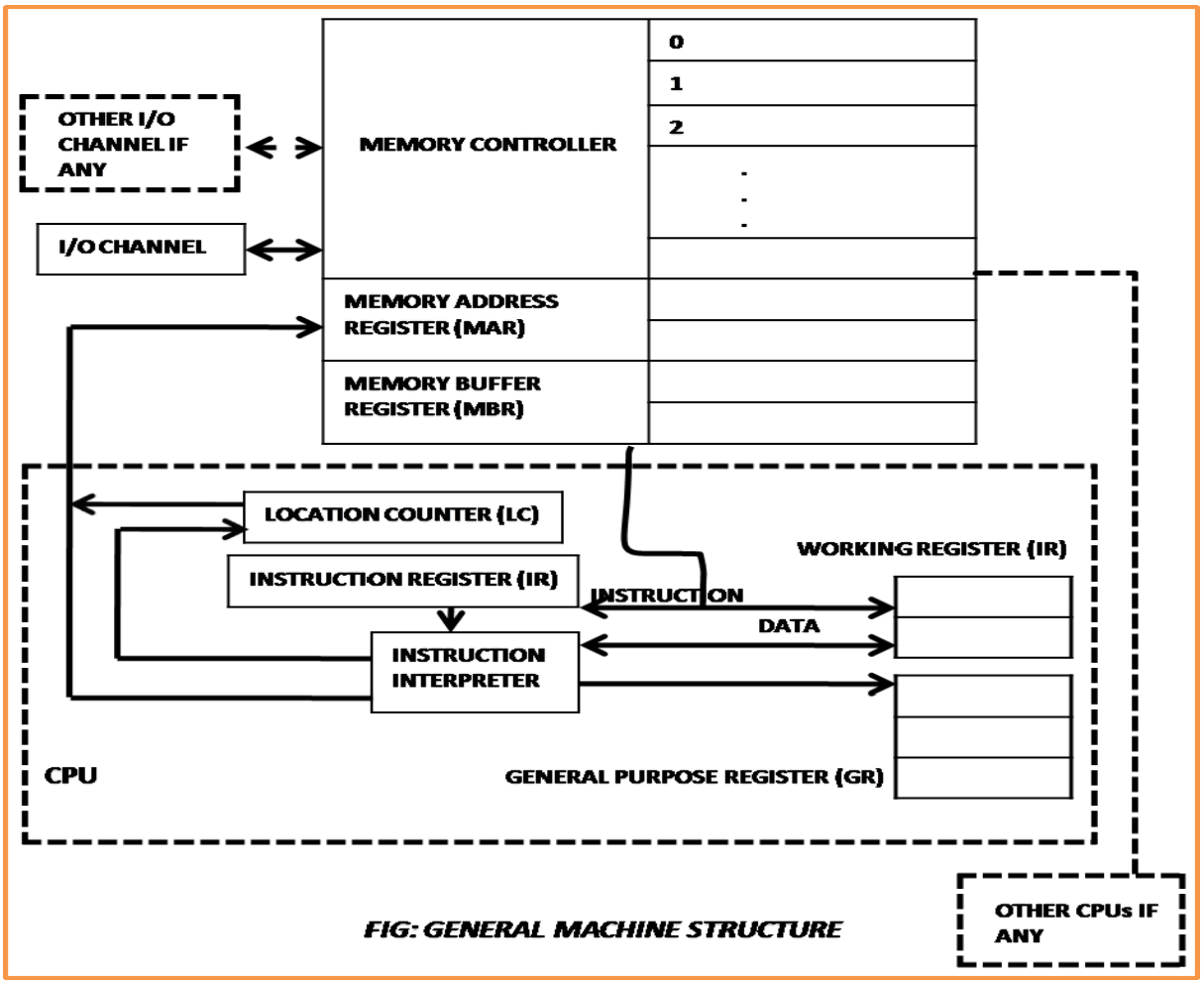


FIG: GENERAL MACHINE STRUCTURE

The components of a general machine are as follows:

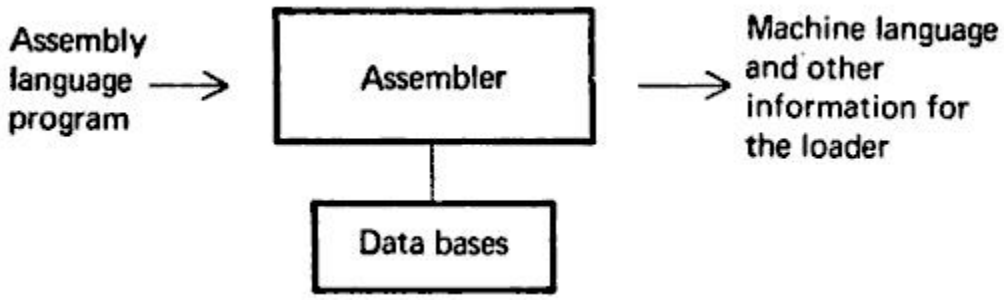
1. **Instruction interpreter:** A group of electronic circuits performs the intent of instruction of fetched from memory.
2. **Location counter:** LC otherwise called as program counter PC or instruction counter IC, is a hardware memory device which denotes the location of the current instruction being executed.
3. **Instruction register:** A copy of the content of the LC is stored in IR.
4. **Working register:** are the memory devices that serve as “scratch pad” for the instruction interpreter.

5. **General register:** are used by programmers as storage locations and for special functions.
6. **Memory address registers (MAR):** contains the address of the memory location that is to read from or stored into.
7. **Memory buffer register (MBR):** contain a copy of the content of the memory location whose address is stored in MAR. The primary interface between the memory and the CPU is through memory buffer register.
8. **Memory controller:** is a hardware device whose work is to transfer the content of the MBR to the core memory location whose address is stored in MAR.
9. **I/O channels:** may be thought of as separate computers which interpret special instructions for inputting and outputting information from the memory.

----*-----**

Assembly Language

- An assembler is a program that takes computer instruction and converts them into a pattern of bits that the computer processor can use to perform its basic operation.
- The assembler is responsible for translating the assembly language program into machine code. When the source program language is essentially a symbolic representation for a numerical machine language, the translator is called assembler and the source language is called an assembly language.



Basic function of Assembler

- Translate mnemonics opcodes to machine language.

- Convert symbolic operands to their machine addresses.
- Build machine instructions in the proper format
- Convert data constants into machine representation.
- Error checking is provided.
- Changes can be quickly and easily incorporated with a reassembly.
- Variables are represented by symbolic names, not as memory locations.
- Assembly language statements are written one per line. A machine code program thus consists of a sequence of assembly language statements, where each statement contains a mnemonics.

Advantages

- Reduced errors
- Faster translation times
- Changes could be made easier and faster.
- Addresses are symbolic, not absolute \
- Easy to remember

Disadvantages

- Assembler language are unique to specific types of computer
- Program is not portable to the computer.
- Many instructions are required to achieve small tasks
- Programmer required knowledge of the processor architecture and instruction set.

Translation phase of Assembler

The six steps that should be followed by the designer

1. Specify the problem
2. Specify data structure
3. Define format of data structure
4. Specify algorithm
5. Look for modularity
6. Repeat 1 through 5 on modules

Functions / Purpose of Assembler

An assembler must do the following

1. Generate instruction
    a. Evaluate the mnemonics in the operation field to produce the machine code
    b. Evaluate the subfield-fine the value of each symbol. Process literals and assign addresses.
2. Process pseudo ops
    a. Pass 1 (Define symbol and literals)
        i. Determine length of machine instruction (MOTGET)
        ii. Keep track of location counter (LC)
        iii. Remember value of symbol until pass 2 (STSTO)
        iv. Process some pseudo ops(POTGET1)
        v. Remember literal (LITSTO)
    b. Pass 2 (Generate object Program)
        i. Look up value of symbol (STGET)
        ii. Generate instruction (MOTGET2)
        iii. Generate data (for DS, DC and Literal)
        iv. Process pseudo ops (POTGET2)

Design data structure for assembler design in Pass-1 and Pass-2 with flow chart

Pass -1

1. Input source program
2. A location counter used to keep track of each instruction location.
3. A table, the machine –operation table (MOT) that indicate the symbolic mnemonics for each instruction and its length (tow, four or six bytes)
4. A table, the pseudo operation table (POT) that indicate the symbolic mnemonics and action to be taken for each pseudo-op in pass-1
5. A table, the literal table (LT) that is used to store each literal encounter and its corresponding assigned location.
6. A table, the symbol table (ST) that is used to store each label and its corresponding value.
7. A copy of the input to be used later by Pass-2. This may be stored in a secondary storage device

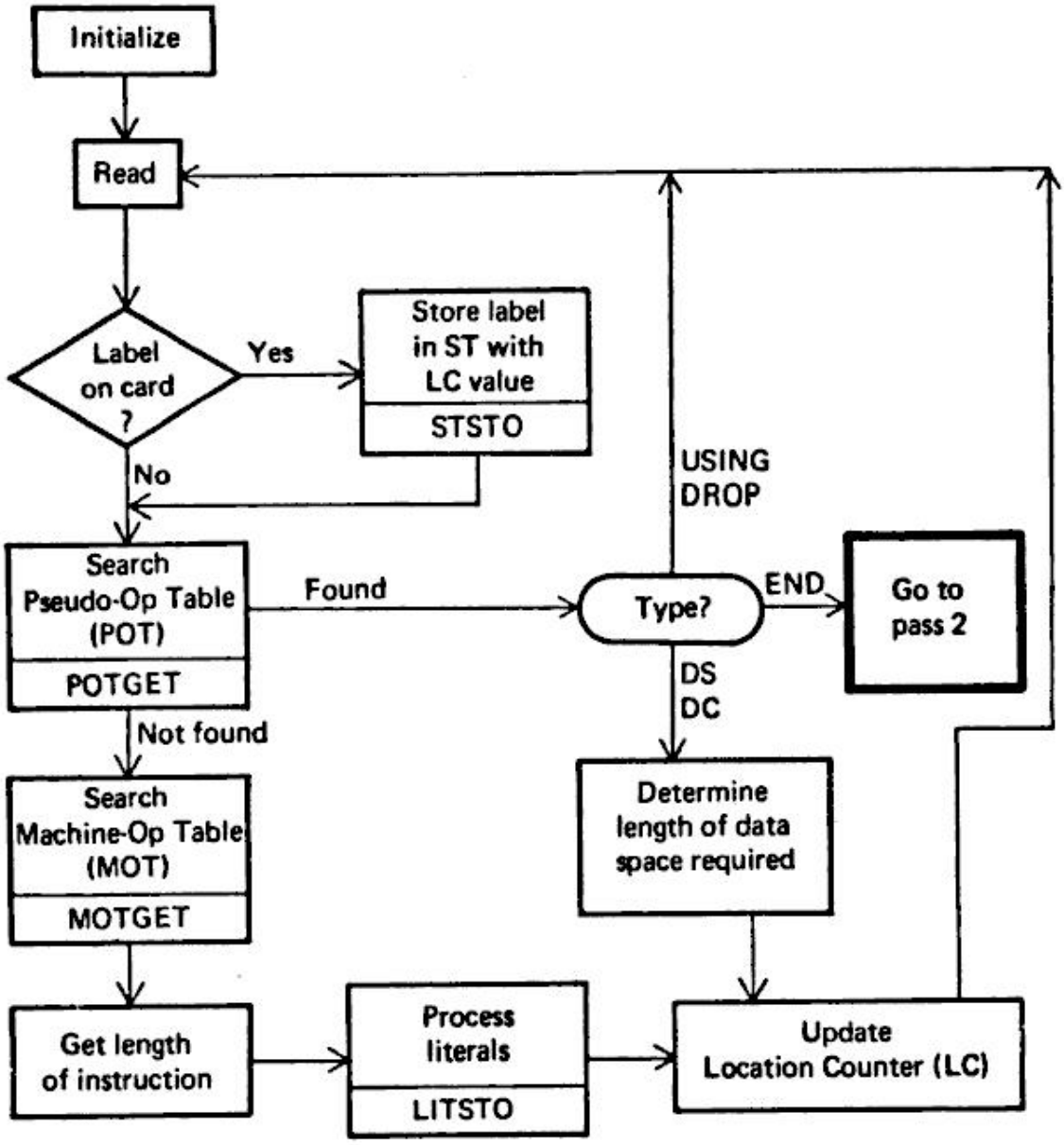


FIGURE 3.3 Pass 1 overview: define symbols

Pass-2

1. Copy of source program input to pass-1
2. Location counter
3. A table the MOT that indicates for each instruction
    a. Symbolic
    b. Mnemonics
    c. Length
    d. Binary machine op-code
    e. Format (RR, RS, RX, SI, SS)

4. A table the POT that indicates for each pseudo-op the symbolic mnemonic and the action to be taken in Pass-2
5. The ST prepare by Pass-1, containing each label and its corresponding value.
6. A table, BT that indicate which register are currently specified by base register by USING pseudo-ops and what are the specified contents of these register.
7. A work space INSR that is to hold each instruction as its various parts are being assembled together.
8. A workspace PRINT LINE used to produce a printed listing
9. A workspace PUNCH CARD used prior to actual outputting for converting assembled instruction into the format needed by the loader.
10. An output deck of assembled instruction in the format needed by the loader.

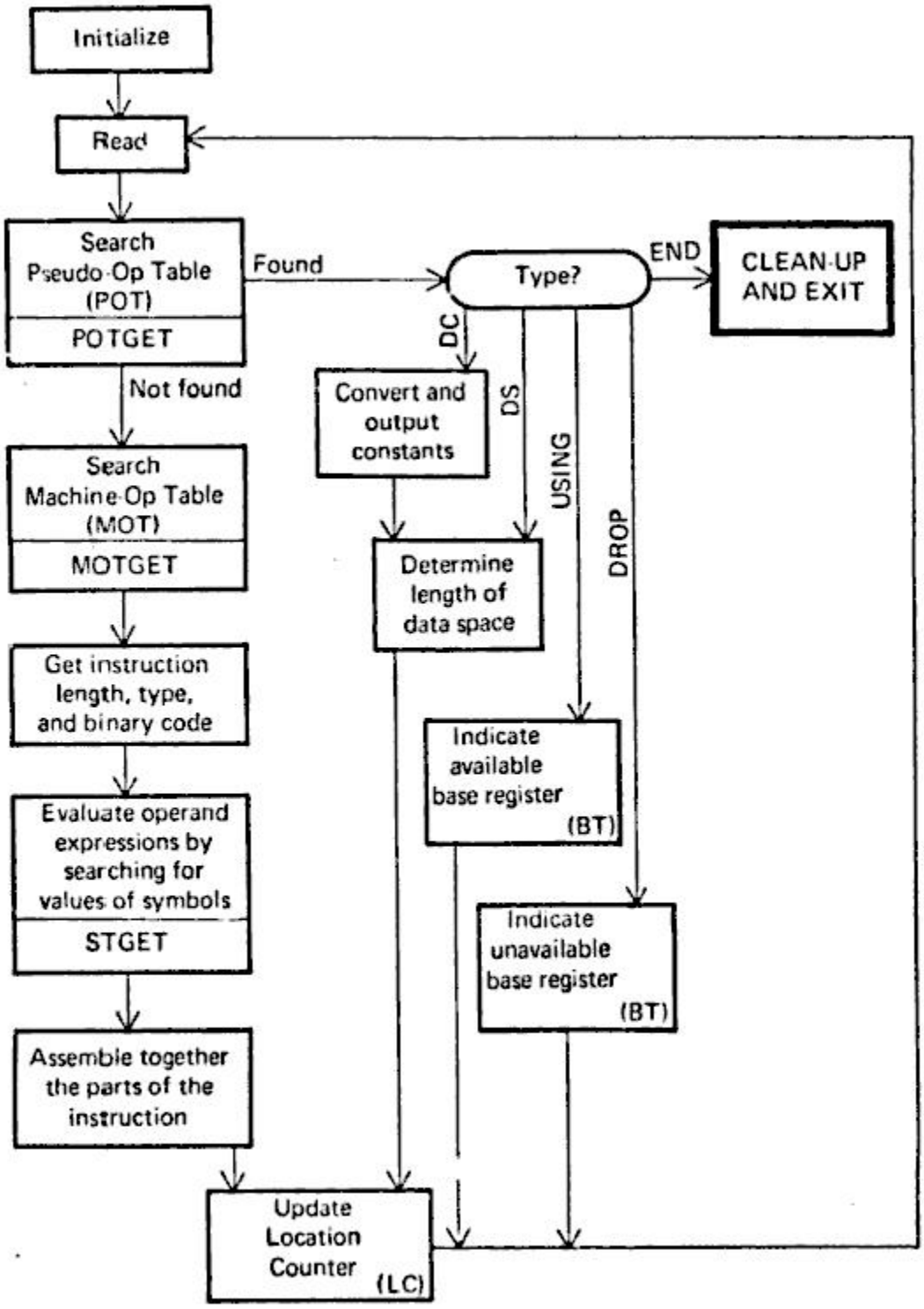


FIGURE 3.4 Pass 2 overview: evaluate fields and generate code

Format of Database

1. Machine Operation Table (MOT)

Fig machine op table for pass-1 and pass-2 the op code is the key and its value is the binary op-code equivalent which is stored for use in generating machine op-code. The instruction length is stored for use in updating the location counter, the instruction format for use in forming the machine language equivalent.

6-bytes per entry

Mnemonic op-code (4-bytes) (characters)	Binary op-code (1-byte) (hexadecimal)	Instruction length (2-bits) (binary)	Instruction format (3-bits) (binary)	Not used in this design (3-bits)
"A*bbb*"	5A	10	001	
"AH*bb*"	4A	10	001	
"AL*bb*"	5E	10	001	
"ALR*b*"	1E	01	000	
"AR*bb*"	1A	01	000	
...	...	...	...	
"MVC*b*"	D2	11	100	
...	...	...	...	

b ~represents the character "blank"

Codes:

Instruction length

01 = 1 half-words = 2 bytes
10 = 2 half-words = 4 bytes
11 = 3 half-words = 6 bytes

Instruction format

000 = RR
001 = RX
010 = RS
011 = SI
100 = SS

2. Pseudo Operation Table (POT)

The table will actually contain the physical address. Fig POT for pass-1, each pseudo-op is listed with and associated pointer to the assembler routine for processing the pseudo-op.

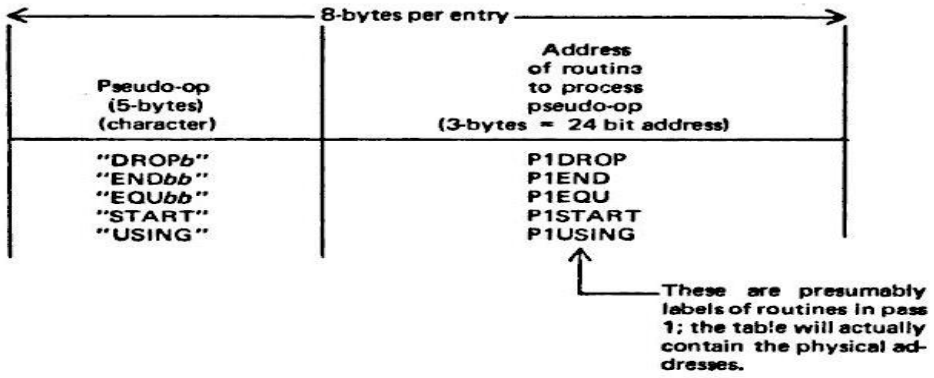


FIGURE 3.7 Pseudo-Op Table (POT) for pass 1 (similar table for pass 2)

3. Symbol Table (ST)

Symbol table is usually hash-organized. It contains all relevant information about symbol defined and used in the source program. Information regarding all forward references in a symbol table, for each symbol and address is generated randomly and the information regarding that symbol get the same address conflict is resolve by using collision handling techniques.

The relative location indicator tells the assembler whether the value of the symbol is absolute or relative base of the program.

14-bytes per entry

Symbol (8-bytes) (characters)	Value (4-bytes) (hexadecimal)	Length (1-byte) (hexadecimal)	Relocation (1-byte) (character)
"JOHN*bbbb*"	0000	01	"R"
"FOUR*bbbb*"	000C	04	"R"
"FIVE*bbbb*"	0010	04	"R"
"TEMP*bbbb*"	0014	04	"R"

FIGURE 3.8 Symbol Table (ST) for pass 1 and pass 2

4. Base Table (BT)

Base table is used by the assembler to generate the proper base register references in machine instruction and to compute the correct offsets, when generating an address, the assembler referencing the base register table to chose a base register that will contains a value close to the symbolic references. The address is the information using that base register.

4-bytes per entry

	Availability indicator (1-byte) (character)	Designated relative-address Contents of base register (3-bytes = 24-bit address) (hexadecimal)
1	"N"	–
2	"N"	–
⋮	⋮	
14	"N"	–
15	"Y"	00 00 00

15 entries

Code=

Availability

Y ~ register specified in USING pseudo-op

N ~ register never specified in USING pseudo-op or subsequently made unavailable by the DROP pseudo-op

FIGURE 3.9 Base Table (BT) for pass 2

Difference Between

Difference between compiler and interpreter

Sr. No.	Compiler	Interpreter
1	Scans the entire program first and then translate it into machine code	Translate the program line-by-line.
2	Convert the entire program to machine code; when all the syntax errors have been removed execution takes place.	Each time the program is executed, every line is checked for syntax error and then converted to equivalent machine code
3	Execution time is less	Execution time is more
4	Machine code can be saved and used; source code and compiler no longer needed.	Machine code cannot be saved; interpreter is always required for translation.
5	Since source code is not required tampering with the source code is not possible	Source code can be easily modified and hence no security of programs.
6	Slow for debugging	Fast for debugging

Difference between process and Programs

Sr. No.	Process	Programs
1	A process is an instance of program in execution	Program is a set of instructions written to carry out a particular task
2	Process is a dynamic concept	Program is a static concept
3	A process is termed as an 'active entity' since it is always stored in the main memory and disappears if the machine is power cycled. Several processes may be associated with a same program.	A program is an executable file residing on the disk (secondary storage) in a directory
4		It is read into the primary memory and executed by the kernel.
5	A **process is the actual execution** of those instructions.	A computer **program is a passive collection** of instructions;

Difference between multiprogramming and multiprocessing

Sr. No.	Multiprocessing	Multiprogramming
1	Multiprocessing refers to processing of multiple processes at same time by multiple CPUs.	Multiprogramming keeps several programs in main memory at the same time and execute them concurrently utilizing single CPU.
2	It utilizes multiple CPUs.	It utilizes single CPU.
3	It permits parallel processing.	Context switching takes place.
4	Less time taken to process the jobs.	More Time taken to process the jobs.
5	It facilitates much efficient utilization of devices of the computer system.	Less efficient than multiprocessing.
6	Usually more expensive.	Such systems are less expensive.

Difference between Open Subroutine and Closed Subroutine

Sr. No.	Open Subroutine	Closed Subroutine
1	Open subroutine is one whose code inserted into the main program	Close subroutine can be stored outside the main routine and control transfer to the subroutine
2	If some open subroutine where called four times. It would appear in four different placed in the calling program	Close subroutine perform transfer of control and transfer of data
3	Arguments are passed in the registers that are given as arguments to the subroutine.	Arguments may be placed in registers or on the stack
4	Open Subroutines are very efficient with no wasted instructions	A subroutine also allows you to debug code once and then ensure that all future instantiations of the code

		will be correct
5	Open Subroutines are very flexible and can be as general as the program wishes to make them	Any register that the subroutine uses must first be **saved** and then **restored** after the subroutine completes execution

Difference between Pure Procedure and Impure Procedure

Sr. No.	Pure Procedure	Impure Procedure
1	A pure procedure does not modify itself.	Procedures that modify themselves are called impure procedures.
2	It can be shared by multiple processors	Other program finds them difficult to read and moreover they cannot shard by multiple processors.
3	Pure procedures are readily reusable.	Impute procedures are not readily reusable.
4	To ensure that the instructions are the same each time a program is used.	Each processor executing an impure procedure modifies its contents.
5	Writing such procedure is a good programming practice	Writing such procedure is a poor programming practice

Difference between user viewpoint and System viewpoint

Operating System is designed both by taking user view and system view into consideration.

1. The goal of the Operating System is to maximize the work and minimize the effort of the user.
2. Most of the systems are designed to be operated by single user, however in some systems multiple users can share resources, memory. In these cases Operating System is designed to handle available resources among multiple users and CPU efficiently.

3. Operating System must be designed by taking both usability and efficient resource utilization into view.
4. In embedded systems (Automated systems) user view is not present.
5. Operating System gives an effect to the user as if the processor is dealing only with the current task, but in background processor is dealing with several processes.

System View

1. From the system point of view Operating System is a program involved with the hardware.
2. Operating System is allocator, which allocate memory, resources among various processes. It controls the sharing of resources among programs.
3. It prevents improper usage, error and handles deadlock conditions.
4. It is a program that runs all the time in the system in the form of Kernel.
5. It controls application programs that are not part of Kernel.

-----***-----

General Approaches to New Machines

In order to know a new machine we have a number of questions in mind. These questions can be categorized as follows.

- **Memory:** Basic unit, size and addressing scheme.
- **Registers** Number of registers, and size, functions, interrelation of each register.
- **Data:** Types of data and their storing scheme.
- **Instruction:** Classes of instructions, allowable operations and their storing scheme.
- **Special Features:** Additional features like interrupt and protections.

The lists of opcodes used in the program are as follows

- **USING** is a pseudo code the indicates to the assembler which general purpose register to use as a base and what its contents will be. As we do not have any specific general register acting as the base register, so it becomes necessary to inform indicate a base register for the program. Because the address are relative so by the knowledge of base and offset the program can be easily be located and executed.

- **BALR** is machine opcode that load a register with the next address and branch to the address in the second field. Since second operand is 0 so the control will go to the next instruction.

- **START** is a pseudo opcode that tell the assembler where the beginning of the program is and allows the user to give a name to the program.

- **END** is a pseudo code that tells the assembler that the last card of the program has been reached.

- **BR 14**, the last machine opcode is used to branch to the location whose address is in general purpose register 14. By convention, calling programs leave their return address in register 14.

Literals

The same program is repeated by using literals, that are mechanisms where by the assembler creates data areas for the programmer, containing constants he requests. =F'10', =F'49', =F'4' are the literal which would be result in the creation of a data area containing 10, 49 and 4 and replacement of the literal operand with the address of the data it describes. L 3, =F'10' is translated by the assembler to point to a full word that contains a 10. Generally the assembler keeps track of the literal with the help of a literal table. This table will contain all the constants that have been requested through the use of literal. A pseudo opcode LTORG place the literal at an earlier location. This is required because, the program may be using 10000 data and it become difficult for the offset of the load instruction to reach the literal at the end of the program.

Data Format of IBM 360/370

The 360 may store several different types of data as is depicted in the figure. The groups of bits stored in memory are interpreted by 360 processor in several ways. The list of different interpretation are shown in the figure are as follow.

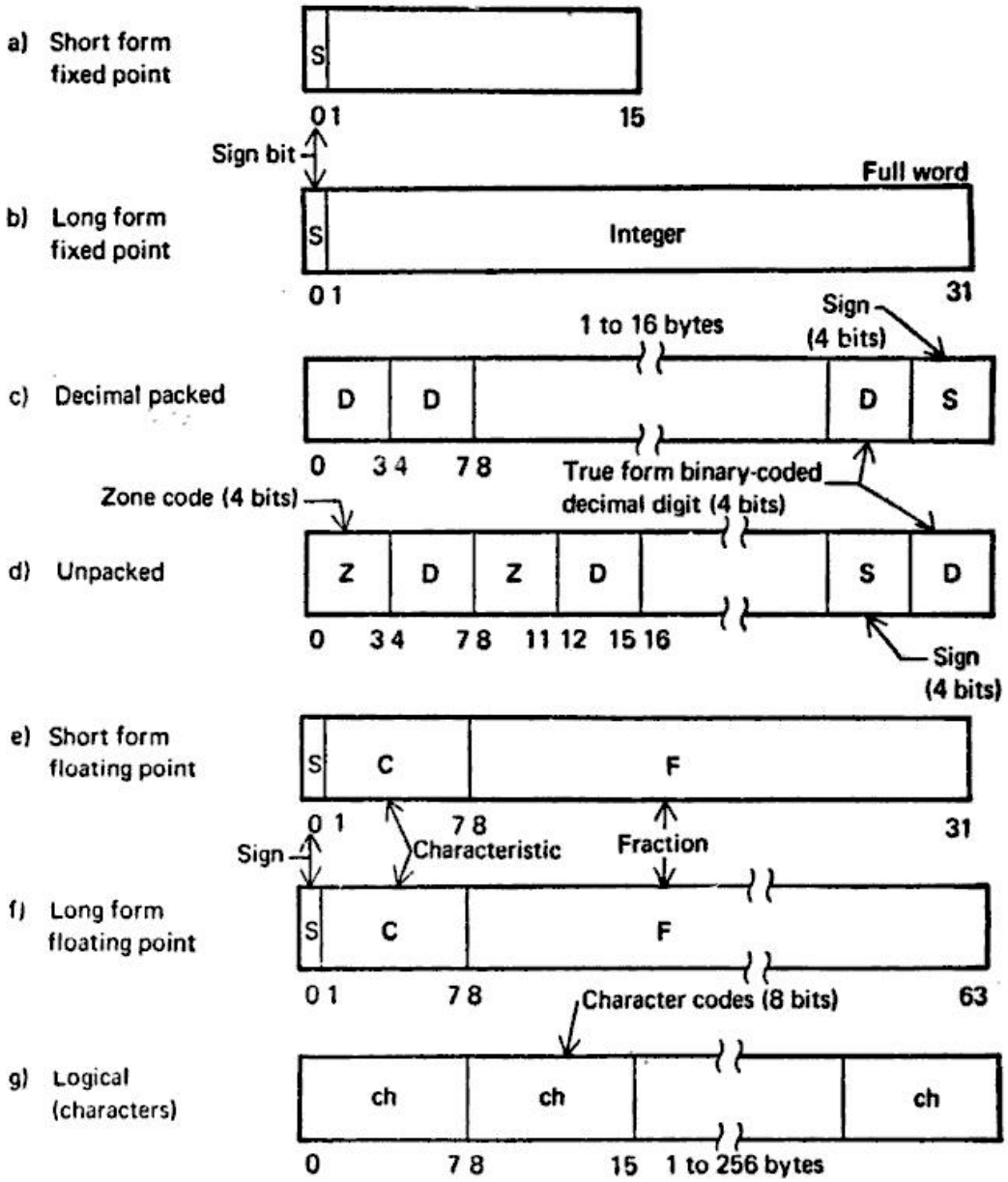
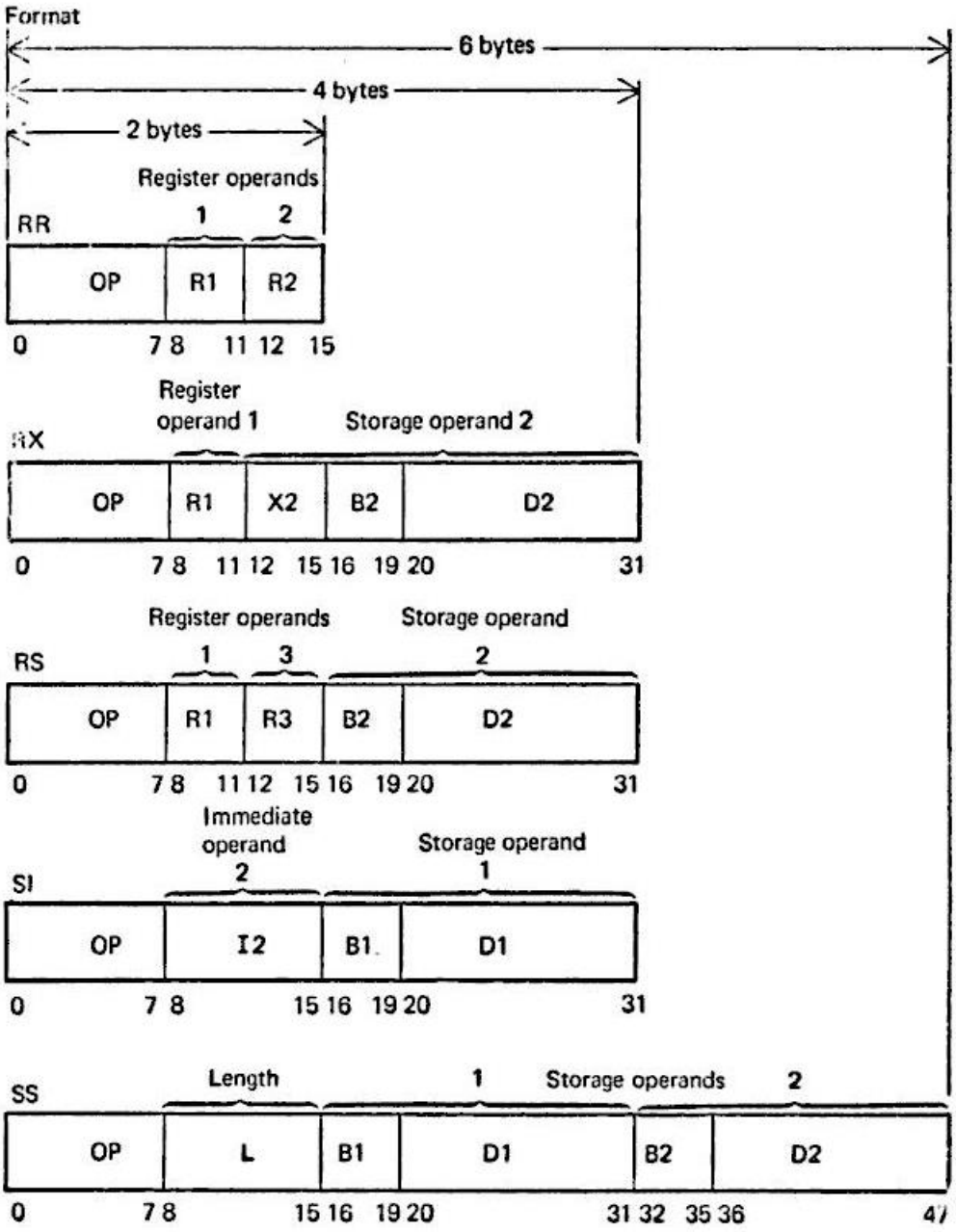


FIGURE 2.3 Data formats for the system/360 and 370

----/////-----

Instruction Format

The instructions in 360 can be arithmetic, logical, control or transfer and special interrupt instructions. The format of 360 instructions is as in figure above.



Mnemonics used:

OP	~	operation code
Ri	~	contents of general register used as operand
Xi	~	contents of general register used as index
Bi	~	contents of general register used as base
Di	~	displacement
Ii	~	immediate data
L	~	operand length

FIGURE 2.5 Basic 360 instruction formats

There are five types of instructions that differ in the type of operands they use.

Register operand refers to the data stored in the 16 general purpose registers (32 bits each). Registers being high-speed circuits provide faster access to data than the data in the core.

E.g. Add register 3, 4 causes the contents of the contents of the register 4 to be added to that of register 3 and stored back in the register 3. The instruction is represented as given in the diagram. The is called as RR format. A total of two bytes are required to represent the RR instruction 8 bits for opcode and 4 bits each of register(8+4+4=16 bits =2 bytes).

The address of ith storage operand is computed from the instruction in the following manner:

Address = c (Bi)+c(Xi)+Di (RX format)

or = c (Bi) +Di (RS, SI, SS format)

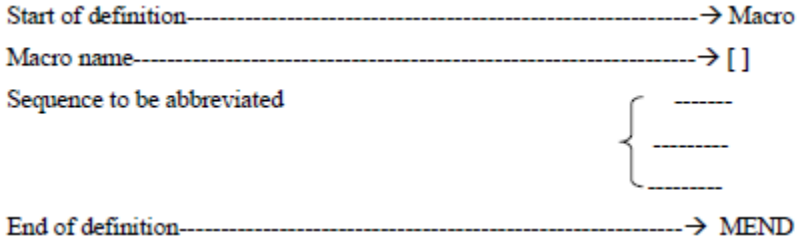
Unit 2

Definition

Macros are single-line abbreviations for a certain group of instructions. Once the macro is defined, these groups of instructions can be used anywhere in a program.

It is sometimes necessary for an assembly language programmer to repeat some blocks of code in the course of a program. The programmer needs to define a single machine instruction to represent a block of code for employing a macro in the program. The macro proves to be useful when instead of writing the entire block again and again, you can simply write the macro that you have already defined. An assembly language macro is an instruction that represents several other machine language instructions at once.

Macro facility permits you to attach a name to the sequence that is occurring several times in a program and then you can easily use this name when that sequence is encountered. All you need to do is to attach a name to a sequence with the help of the macro instruction definition. The following structure shows how to define a macro in a program:



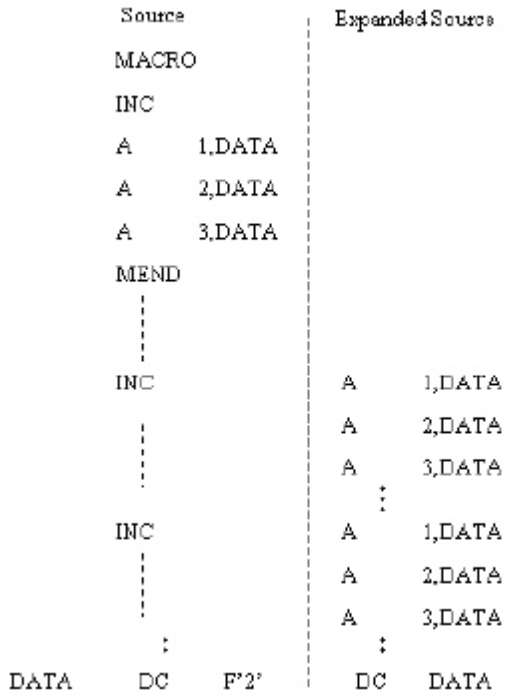
This structure describes the macro definition in which the first line of the definition is the MACRO pseudo-op. Following line is the name line for macro, which identifies the macro

instruction name. The line following the macro name includes the sequence of instructions that are being abbreviated. Each instruction comprises of the actual macro instruction. The last statement in the macro definition is MEND pseudo-op. This pseudo-op denotes the end of the macro definition and terminates the definition of macro instruction.

MACRO EXPANSION

Once a macro is being created, the interpreter or compiler automatically replaces the pattern, described in the macro, when it is encountered. The macro expansion always happens at the compile-time in compiled languages. The tool that performs the macro expansion is known as macro expander. Once a macro is defined, the macro name can be used instead of using the entire instruction sequence again and again.

As you need not write the entire program repeatedly while expanding macros, the overhead associated with macros is very less. This can be explained with the help of the following example.



The macro processor replaces each macro call with the following lines:

```
A    1,    DATA
A    2,    DATA
A    3,    DATA
```

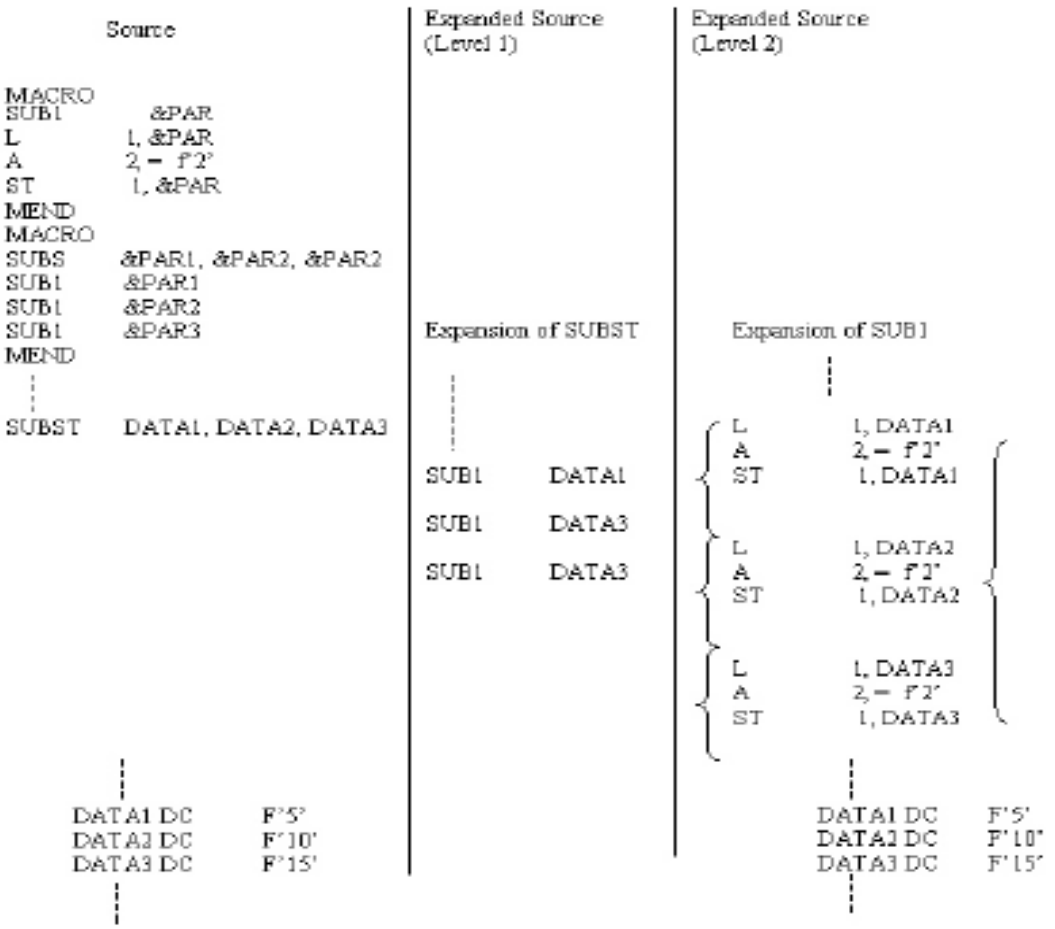
The process of such a replacement is known as expanding the macro. The macro definition itself does not appear in the expanded source code. This is because the macro processor saves the definition of the macro. In addition, the occurrence of the macro name in the source program refers to a macro call. When the macro is called in the program, the sequence of instructions corresponding to that macro name gets replaced in the expanded source.

NESTED MACRO CALLS

Nested macro calls refer to the macro calls within the macros. A macro is available within other macro definitions also. In the scenario where a macro call occurs, which contains another macro call; the macro processor generates the nested macro definition as text and places it on the input stack. The definition of the macro is then scanned and the macro processor compiles it. This is important to note that the macro call is nested and not the macro definition. If you nest the macro definition, the macro processor compiles the same macro repeatedly, whenever the section of the outer macro is executed. The following example can make you understand the nested macro calls:

```
MACRO
SUB 1  &PAR
L      1, & PAR
A      1, = F'2'
ST     1, &PAR
MEND
MACRO
SUBST   &PAR1, &PAR2, &PAR3
SUB1   &PAR1
SUB1    &PAR2
SUB1    &PAR3
2MEND
```

You can easily notice from this example that the definition of the macro 'SUBST' contains three separate calls to a previously defined macro 'SUB1'. The definition of the macro SUB1 has shortened the length of the definition of the macro 'SUBST'. Although this technique makes the program easier to understand, at the same time, it is considered as an inefficient technique. This technique uses several macros that result in macro expansions on multiple levels.



This is clear from the example that a macro call, SUBST, in the source is expanded in the expanded source (Level 1) with the help of SUB1, which is further expanded in the expanded source (Level 2).

FEATURES OF MACRO FACILITY

The features of the macro facility are as follows:

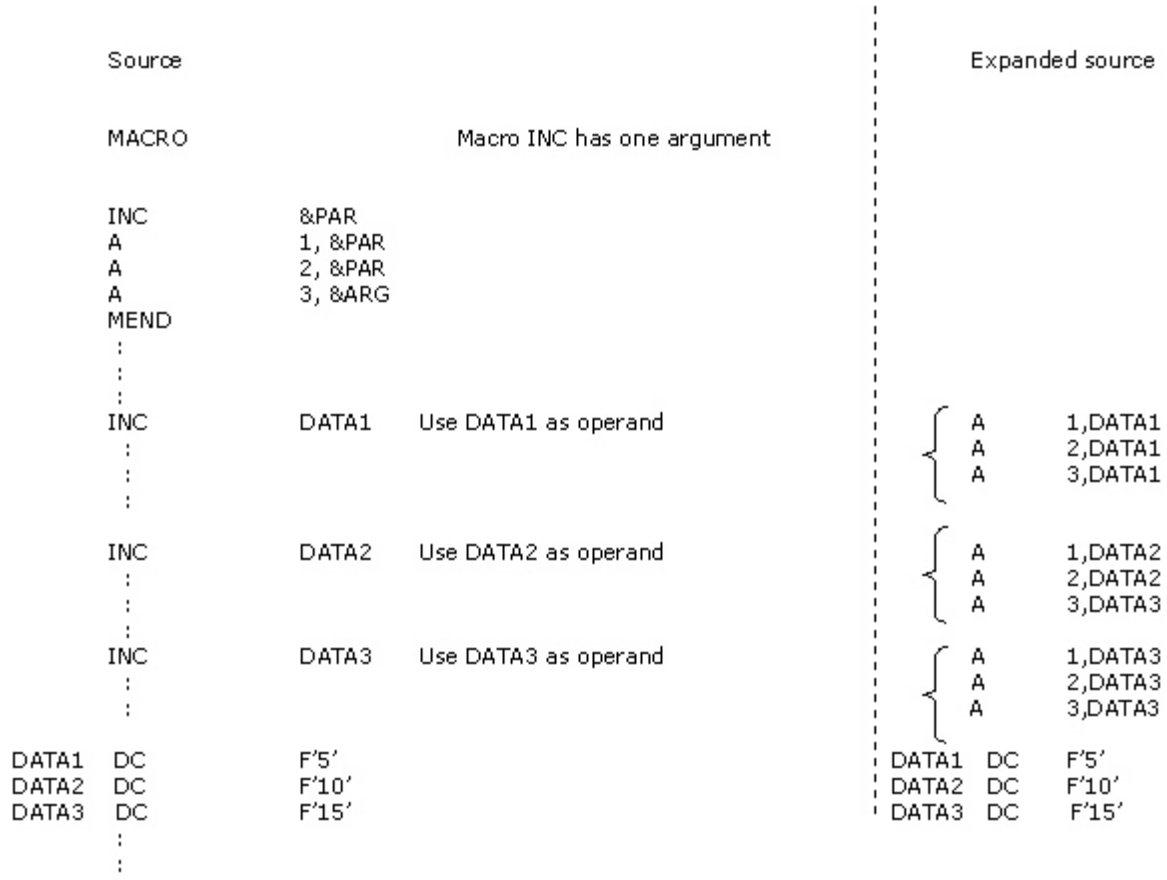
- Macro instruction arguments
- Conditional macro expansion
- Macro instructions defining macros

1. Macro Instruction Arguments

The macro facility presented so far inserts block of instructions in place of macro calls. This facility is not at all flexible, in terms that you cannot modify the coding of the macro name for a specific macro call. An important extension of this facility consists of providing the arguments or parameters in the macro calls. Consider the following program.

```
        :
        :
        :
A       1,DATA1
A       2, DATA1
A       3, DATA1
        :
        :
        :
A       1,DATA2
A       2,DATA2
A       3,DATA2
        :
        :
        :
A       1,DATA3
A       2,DATA3
A       3,DATA3
        DATA1 DC    F'5'
        DATA2 DC    F'10'
        DATA3 DC    F'15'
```

In this example, the instruction sequences are very much similar but these sequences are not identical. It is important to note that the first sequence performs an operation on an operand DATA1. On the other hand, in the second sequence the operation is being performed on operand DATA2. The third sequence performs operations on DATA3. They can be considered to perform the same operation with a variable parameter or argument. This parameter is known as a macro instruction argument or dummy argument.



Notice that in this program, a dummy argument is specified on the macro name line and is distinguished by inserting an ampersand (&) symbol at the beginning of the name. There is no limitation on supplying arguments in a macro call. The important thing to understand about the macro instruction argument is that each argument must correspond to a definition or dummy argument on the macro name line of the macro definition. The supplied arguments are substituted for the respective dummy arguments in the macro definition whenever a macro call is processed.

2. Conditional Macro Expansion

These macro expansions permit conditional reordering of the sequence of macro expansion. They are responsible for the selection of the instructions that appear in the expansions of a macro call. These selections are based on the conditions specified in a program. Branches and tests in the macro instructions permit the use of macros that can be used for assembling the instructions. The facility for selective assembly of these macros is considered as the most powerful programming tool for the system software. The use of the conditional macro expansion can be explained with the help of an example.

Consider the following set of instructions:

```
LOOP 1      A     1,     DATA1
            A     2,     DATA2
            A     3,     DATA3
                  :
                  :
LOOP 2      A     1,     DATA3
```

```
A       2,      DATA2
        :
        :
DATA1           DC F'5'
DATA2           DC F'10'
DATA3           DC F'15'
```

In this example, the operands, labels and number of instructions generated are different in each sequence. Rewriting the set of instructions in a program might look like:

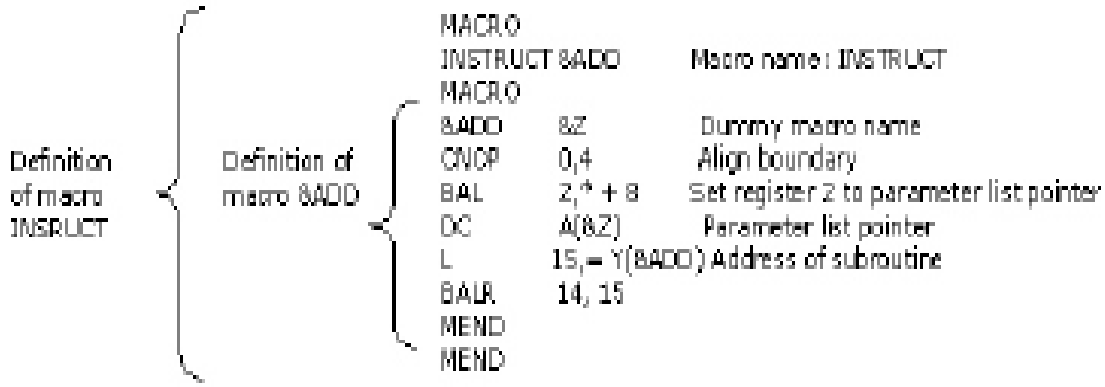
```
                :
                MACRO
&PAR0   VARY            &COUNT, &PAR1, &PAR2, &PAR3
                A               1, &PAR1
                AIF             (&COUNT EQ 1). FINI          Test if &COUNT = 1
                A               2, &PAR2
                AIF             (&COUNT EQ 2). FINI          Test if &COUNT = 2
                ADD             3,&PAR3
.FINI           MEND
                 :                                           Expanded source
                 :
LOOP1   VARY            3,DATA1, DATA2, DATA3        LOOP1   A       1,DATA1
                                                             A       2,DATA2
                :                                            A       3,DATA3
                :
LOOP2   VARY            2,DATA3, DATA2               LOOP2   A       1,DATA3
                :                                            A       2,DATA2
                :

DATA1   DC              F'5'
DATA2   DC              F'10'
DATA3   DC              F'15'
```

The labels starting with a period (.) such as .FINI are macro labels. These macro labels do not appear in the output of the macro processor. The statement AIF (& COUNT EQ 1).FINI directs the macro processor to skip to the statement labeled .FINI, if the parameter corresponding to &COUNT is one. Otherwise, the macro processor continues with the statement that follows the AIF pseudo-op. AIF pseudo-op performs an arithmetic test and since it is a conditional branch pseudo-op, it branches only if the tested condition is true. Another pseudo-op used in this program is AGO, which is an unconditional branch pseudo-op and works as a GOTO statement. This is the label in the macro instruction definition that specifies the sequential processing of instructions from the location where it appears in the instruction. These statements are indications or directives to the macro processor that do not appear in the macro expansions.

3. Macro Instructions Defining Macros

A single macro instruction can also simplify the process of defining a group of similar macros. The considerable idea while using macro instructions defining macros is that the inner macro definition should not be defined until the outer macro has been called once. Consider a macro instruction INSTRUCT in which another subroutine &ADD is also defined.

This is explained in the following macro instruction.



In this code, first the macro INSTRUCT has been defined and then within INSTRUCT, a new macro &ADD is being defined. Macro definitions within macros are also known as "macro definitions within macro definitions".

DESIGN OF A MACRO PRE-PROCESSOR

A Macro pre-processor effectively constitutes a separate language processor with its own language. A macro pre-processor is not really a macro processor, but is considered as a macro translator. The approach of using macro pre-processor simplifies the design and implementation of macro pre-processor. Moreover, this approach can also use the features of macros such as macro calls within macros and recursive macros. Macro pre-processor recognises only the macro definitions that are provided within macros. The macro calls are not considered here because the macro pre-processor does not perform any macro expansion.

The macro preprocessor generally works in two modes: passive and active. The passive mode looks for the macro definitions in the input and copies macro definitions found in the input to the output. By default, the macro pre-processor works in the passive mode. The macro pre-processor switches over to the active mode whenever it finds a macro definition in the input. In this mode, the macro preprocessor is responsible for storing the macro definitions in the internal data structures. When the macro definition is completed and the macros get translated, then the macro pre-processor switches back to the passive mode.

Four basic tasks that are required while specifying the problem in the macro pre-processor are as follows:

1. **Recognising macro definitions**: A macro pre-processor must recognize macro definitions that are identified by the MACRO and MEND pseudo-ops. The macro definitions can be easily recognised, but this task is complicated in cases where the macro definitions appear within macros. In such situations, the macro pre-processor must recognise the nesting and correctly matches the last MEND with the first MACRO.

2. **Saving the definitions**: The pre-processor must save the macro instructions definitions that can be later required for expanding macro calls.

3. **Recognising macro calls**: The pre-processor must recognise macro calls along with the macro definitions. The macro calls appear as operation mnemonics in a program.

4. **Replacing macro definitions with macro calls**: The pre-processor needs to expand macro calls and substitute arguments when any macro call is encountered. The pre-processor must substitute macro definition arguments within a macro call.

Implementation of Two-Pass Algorithm

The two-pass algorithm to design macro pre-processor processes input data into two passes. In first pass, algorithm handles the definition of the macro and in second pass; it handles various calls for macro. Both the passes of two-pass algorithm in detail are:

1. First Pass

The first pass processes the definition of the macro by checking each operation code of the macro. In first pass, each operation code is saved in a table called Macro Definition Table (MDT). Another table is also maintained in first pass called Macro Name Table (MNT). First pass uses various other databases such as Macro Name Table Counter (MNTC) and Macro Name Table Counter (MDTC). The various databases used by first pass are:

1. The input macro source deck.
2. The output macro source deck copies that can be used by pass 2.
3. The Macro Definition Table (MDT), which can be used to store the body of the macro definitions. MDT contains text lines and every line of each macro definition, except the MACRO line gets stored in this table. For example, consider the code described in macro expansion section where macro INC used the macro definition of INC in MDT. Table 2.1 shows the MDT entry for INC macro:

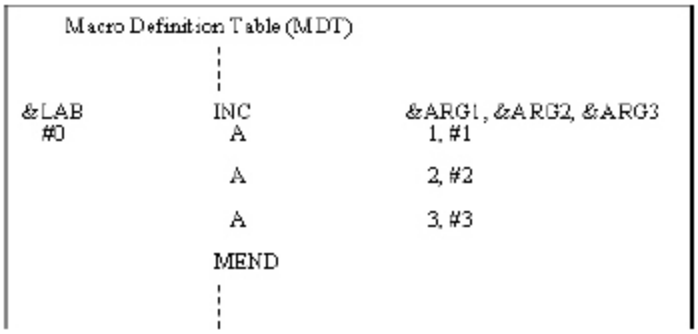
Table 2.1: *MDT*

Macro Definition Table (MDT)

&LAB	INC	&ARG1, &ARG2, &ARG3
#0	A	1, #1
	A	2, #2
	A	3, #3
	MEND	

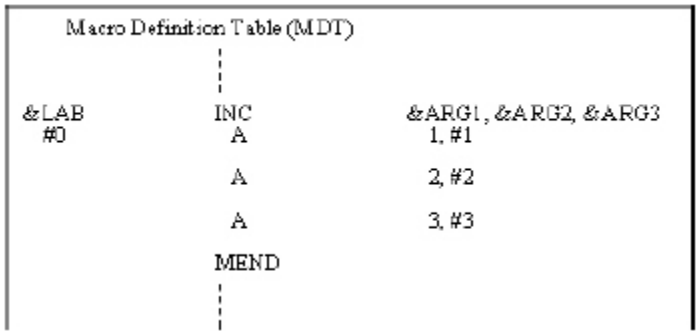
4. The Macro Name Table (MNT), which can be used to store the names of defined macros. Each MNT entry consists of a character string such as the macro name and a pointer such as index to the entry in MDT that corresponds to the beginning of the macro definition. Table 2.2 shows the MNT entry for INCR macro:

Table 2.2: *MNT*



5. The Macro Definition Table Counter (MDTC) that indicates the next available entry in the MDT.
6. The Macro Name Table Counter (MNTC) that indicates the next available entry in the MNT.
7. The Argument List Array (ALA) that can be used to substitute index markers for dummy arguments prior to store a macro definition. ALA is used during both the passes of the macro pre-processor. During Pass 1, dummy arguments in the macro definition are replaced with positional indicators when the macro definition is stored. These positional indicators are used to refer to the memory address in the macro expansion. It is done in order to simplify the later argument replacement during macro expansion. The i_{th} dummy argument on the macro name card is represented in the body of the macro by the index marker symbol #. The # symbol is a symbol reserved for the use of macro pre-processor.

Table 2.3: *ALA*



2. Second Pass

Second pass of two-pass algorithm examine each operation mnemonic such that it replaces macro name with the macro definition. The various data-bases used by second pass are:

1. The copy of the input macro source deck.
2. The output expanded source deck that can be used as an input to then assembler.
3. The MDT that was created by pass 1.
4. The MNT that was created by pass 1.
5. The MDTP for indicating the next line of text that is to be used during macro expansion.
6. The ALA that is used to substitute macro calls arguments for the index markers in the stored macro definition.

3 .Two-Pass Algorithm

In two-pass macro-preprocessor, you have two algorithms to implement, first pass and second pass. Both the algorithms examines line by line over the input data available. Two algorithms to implement two-pass macro-preprocessor are:

- Pass 1 Macro Definition
- Pass 2 Macro Calls and Expansion

Pass 1 - Macro Definition

Pass 1 algorithm examines each line of the input data for macro pseudo opcode. Following are the steps that are performed during Pass 1 algorithm:

1. Initialize MDTC and MNTC with value one, so that previous value of MDTC and MNTC is set to value one.
2. Read the first input data.
3. If this data contains MACRO pseudo opcode then
   A. Read the next data input.
   B. Enter the name of the macro and current value of MDTC in MNT.
   C. Increase the counter value of MNT by value one.
   D. Prepare that argument list array respective to the macro found.
   E. Enter the macro definition into MDT. Increase the counter of MDT by value one.
   F. Read next line of the input data.
   G. Substitute the index notations for dummy arguments passed in macro.
   H. Increase the counter of the MDT by value one.
   I. If mend pseudo opcode is encountered then next source of input data is read.
   J. Else expands data input.
4. If macro pseudo opcode is not encountered in data input then
   A. A copy of input data is created.
   B. If end pseudo opcode is found then go to Pass 2.
   C. Otherwise read next source of input data.

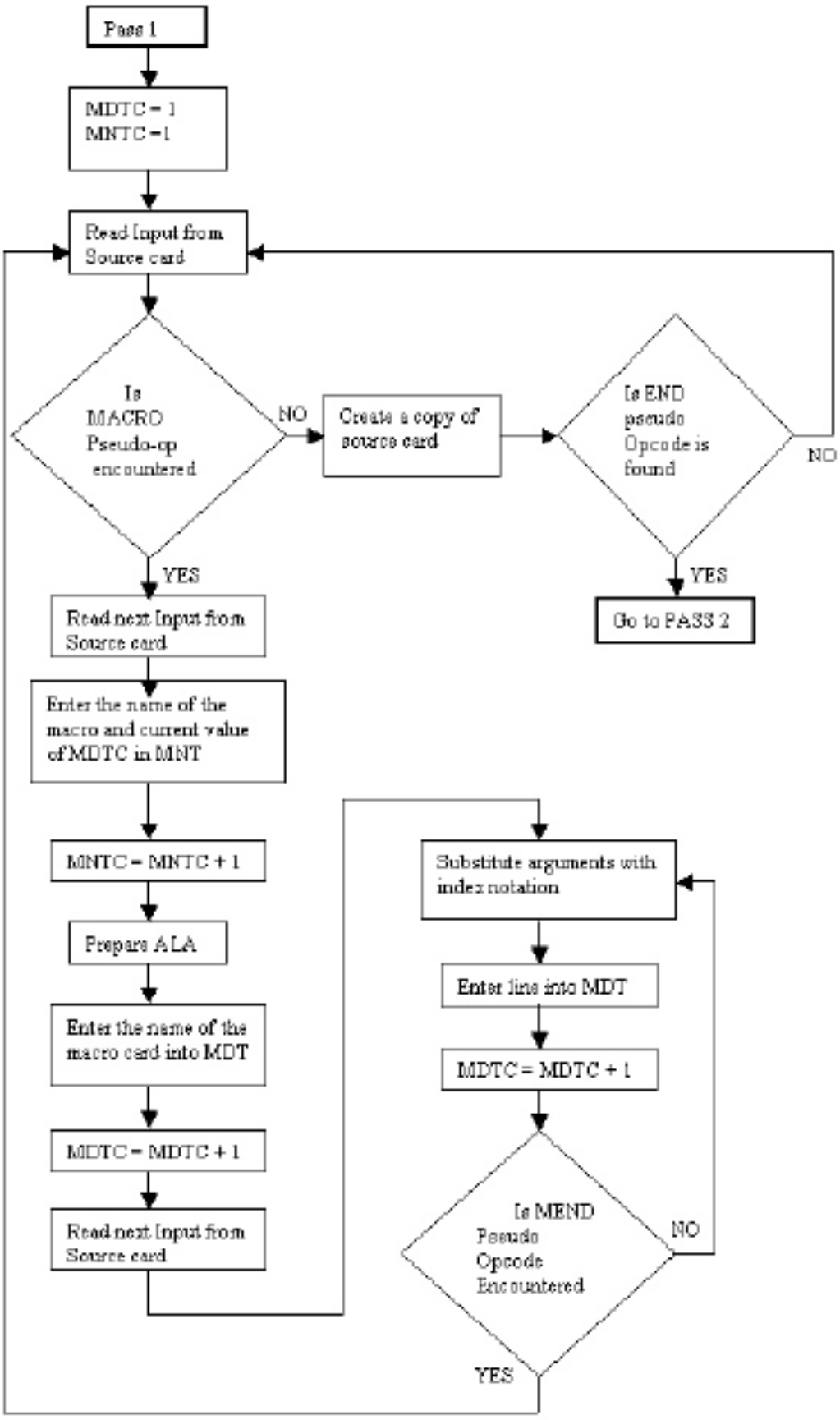


Figure 2.1: Flow Chart for First Pass

Pass 2 - Macro Calls and Expansion

Pass two algorithm examines the operation code of every input line to check whether it exist in MNT or not. Following are the steps that are performed during second pass algorithm:

1. Read the input data received from Pass 1.
2. Examine each operation code for finding respective entry in the MNT.
3. If name of the macro is encountered then
    A. A Pointer is set to the MNT entry where name of the macro is found. This pointer is called
        Macro Definition Table Pointer (MDTP).
    B. Prepare argument list array containing a table of dummy arguments.
    C. Increase the value of MDTP by value one.
    D. Read next line from MDT.
    E. Substitute the values from the arguments list of the macro for dummy arguments.
    F. If mend pseudo opcode is found then next source of input data is read.
    G. Else expands data input.
4. When macro name is not found then create expanded data file.
5. If end pseudo opcode is encountered then feed the expanded source file to assembler for processing.
6. Else read next source of data input.

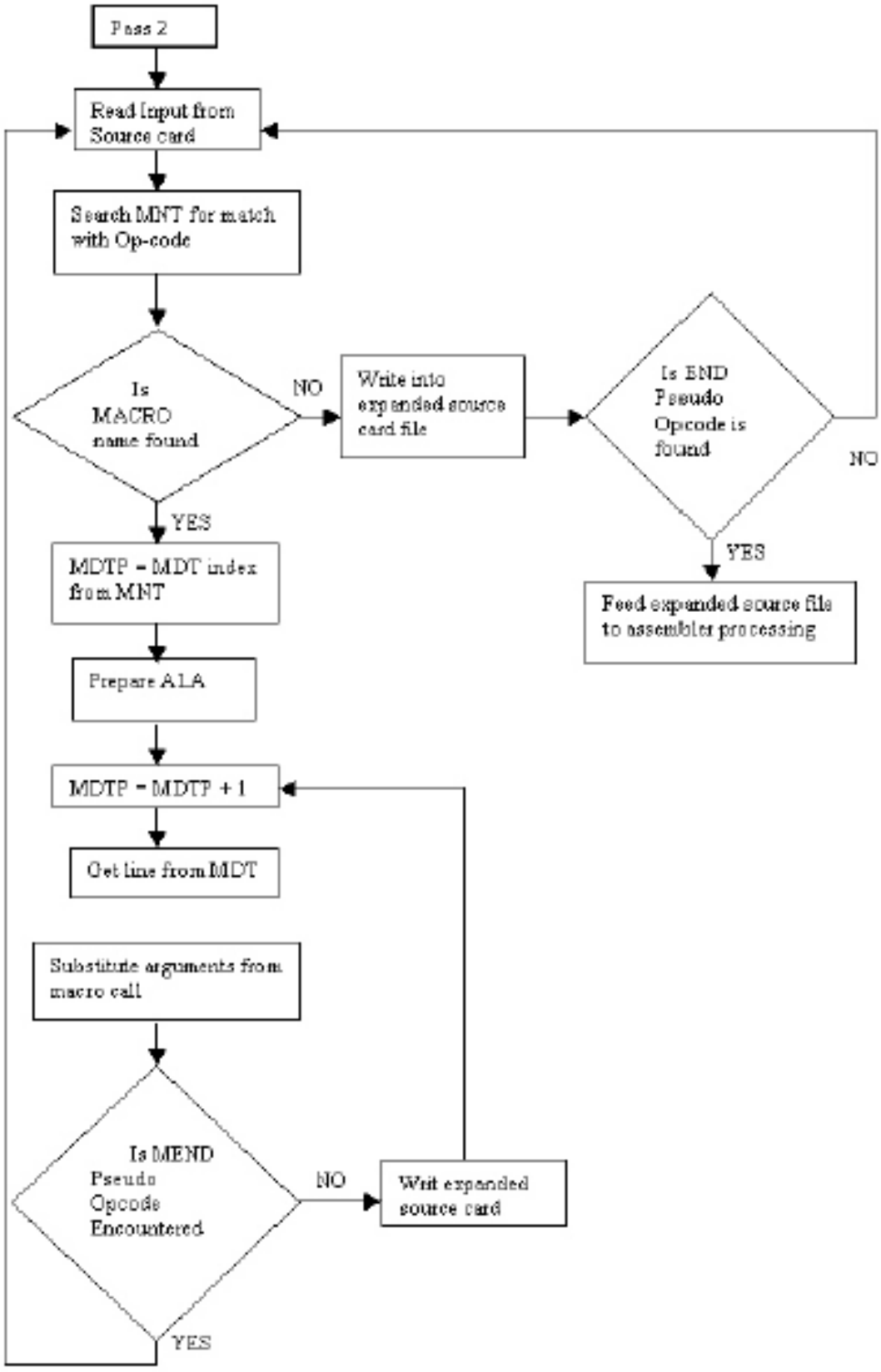


Figure 2.2: Flow Chart for Second Pass

Implementation of Single-Pass Algorithm

The single-pass algorithm allows you to define macro within the macro but not supports macro calls within the macro. In the single-pass algorithm two additional Level Counter (MDLC). Following is the usage of MDI and MDLC in single-pass algorithm:

- **MDI indicator**: Allows you to keep track of macro calls and macro definitions. During expansion of macro call, MDI indicator has value ON and retains value OFF otherwise. If MDI indicator is on, then input data lines are read from MDT until mend pseudo opcode is not encountered. When MDI is off, then data input is read from data source instead of MDI.
- **MDLC indicator**: MDLC ensures you that macro definition is stored in MDT. MDLC is counters that keeps track of the numbers of macro1 and mend pseudo opcode found. Single-pass algorithm combines both the algorithms defined above to implement two-pass macro pre-processor. Following are the steps that are followed during single-pass algorithm:

1. Initialize MDTC and MNTC to value one and MDLC to zero.
2. Set MDI to value OFF.
3. Performs read operation.
4. Examine MNT to get the match with operation code.
5. If macro name is found then
   A. MDI is set to ON.
   B. Prepare argument list array containing a table of dummy arguments.
   C. Performs read operation.
6. Else it examines that macro pseudo opcode is encountered. If macro pseudo opcode is found then
   A. Enter the name of the macro and current value of MDTC in MNT at entry number MNTC.
   B. Increment the MNTC to value one.
   C. Prepare argument list array containing a table of dummy arguments..
   D. Enter the macro card into MDT.
   E. Increment the MDTC to value one.
   F. Increment the MDLC to value one.
   G. Performs read operation.
   H. Substitute the index notations for the arguments list of the macro for dummy arguments.
   I. Enter data input line into MDT.
   J. Increment the MDTC to value one.
   K. If macro pseudo opcode is found then increments the MDLC to value one and performs read
      operation.

L. Else it checks for mend pseudo opcode if not found then performs read operation.

M. If mend pseudo opcode is found then decrement the MDLC to value one.

N. If MDLC is equal to zero then it goes to step 2. Otherwise, it performs read operation.

7. In case macro pseudo opcode is not found, then write it into expanded source card file.

If end pseudo opcode is found, then it feeds expanded source file to assembler for processing, otherwise performs read operation at step 2.

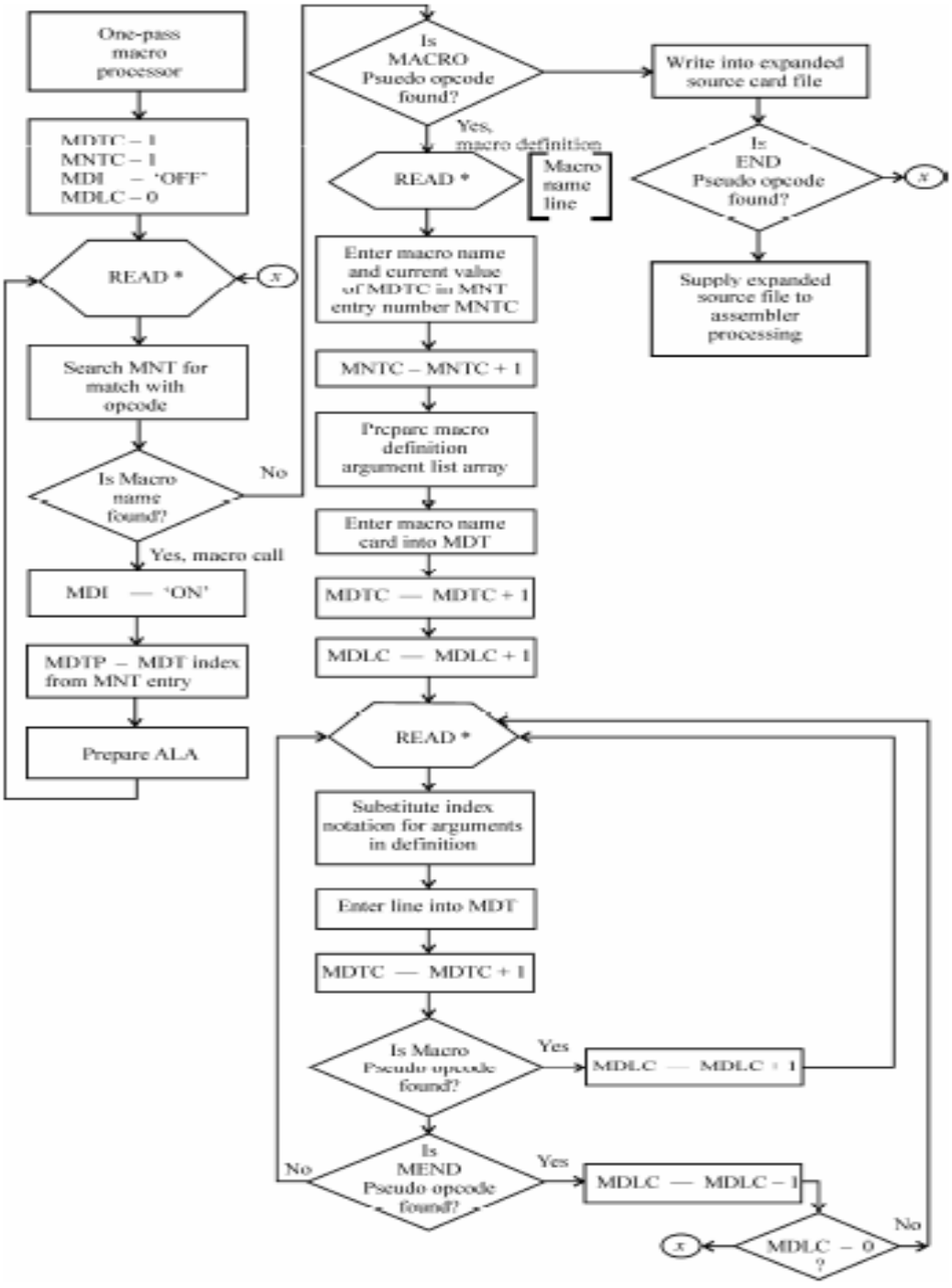


Figure 2.3: Single Pass

Unit 3

INTRODUCTION

Earlier programmers used loaders that could take program routines stored in tapes and combine and relocate them in one program. Later, these loaders evolved into linkage editor used for linking the program routines but the program memory remained expensive and computers were slow. A little progress in linking technology helped computers become faster and disks larger, thus program linking became easier. For the easier use of memory space and efficiency in speed, you need to use linkers and loaders.

Loaders and linker's helps you to have a schematic flow of steps that you need to follow while creating a program. Following are the steps that you need to perform when you write a program in language:

1. Translation of the program, which is performed by a processor called translator.
2. Linking of the program with other programs for execution, this is performed by a separate processor known as linker.
3. Relocation of the program to execute from the memory location allocated to it, which is performed by a processor called loader.
4. Loading of the program in the memory for its execution, this is performed by a loader.

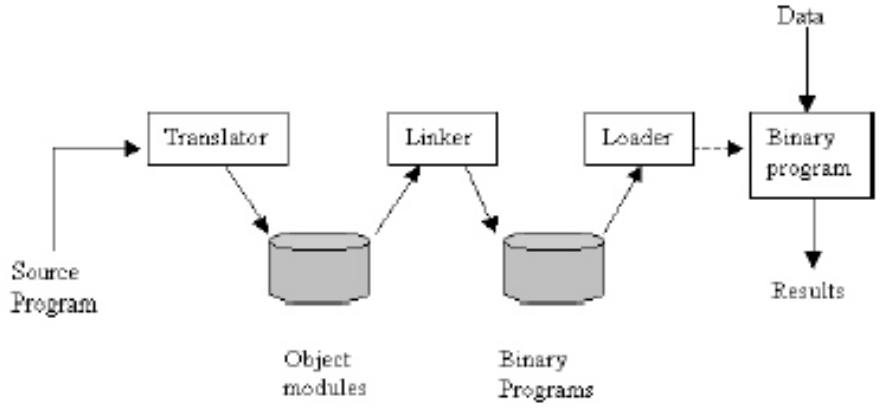


Figure 4.1: Schematic Flow of Program Execution

LOADERS

A loader is a program that performs the functions of a linker program and then immediately schedules the resulting executable program for some kind of action. In other words, a loader accepts the object program, prepares these programs for execution by the computer and then initiates the execution. It is not necessary for the loader to save a program as an executable file.

The functions performed by a loader are as follows:

1. **Memory Allocation** allocates space in memory for the program.

2. **Linking**: **Resolves** symbolic references between the different objects.
3. **Relocation** adjusts all the address dependent locations such as address constants, in order to correspond to the allocated space.
4. **Loading** places the instructions and data into memory.

Functions of Loader:

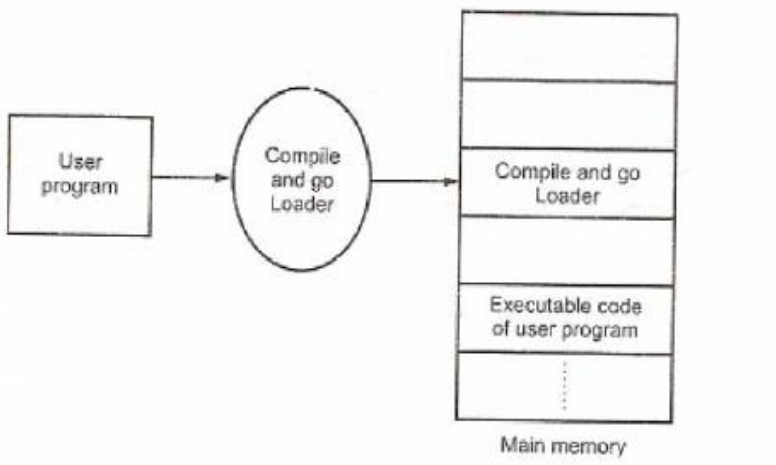
The loader is responsible for the activities such as allocation, linking, relocation and loading

1. It allocates the space for program in the memory, by calculating the size of the program. This activity is called allocation.
2. It resolves the symbolic references (code/data) between the object modules by assigning all the user subroutine and library subroutine addresses. This activity is called linking.
3. There are some address dependent locations in the program, such address constants must be adjusted according to allocated space, such activity done by loader is called relocation.
4. Finally it places all the machine instructions and data of corresponding programs and subroutines into the memory. Thus program now becomes ready for execution, this activity is called loading.

1. Compile and Go Loader

Compile and go loader is also known as "assembler-and-go". It is required to introduce the term "segment" to understand the different loader schemes. A segment is a unit of information such as a program or data that is treated as an entity and corresponds to a single source or object deck. A figure shows the compile and go loader.

The compile and go loader executes the assembler program in one part of memory and places the assembled machine instructions and data directly into their assigned memory locations. Once the assembly is completed, the assembler transfers the control to the starting instruction of the program.



Compile and go loading scheme

Advantages

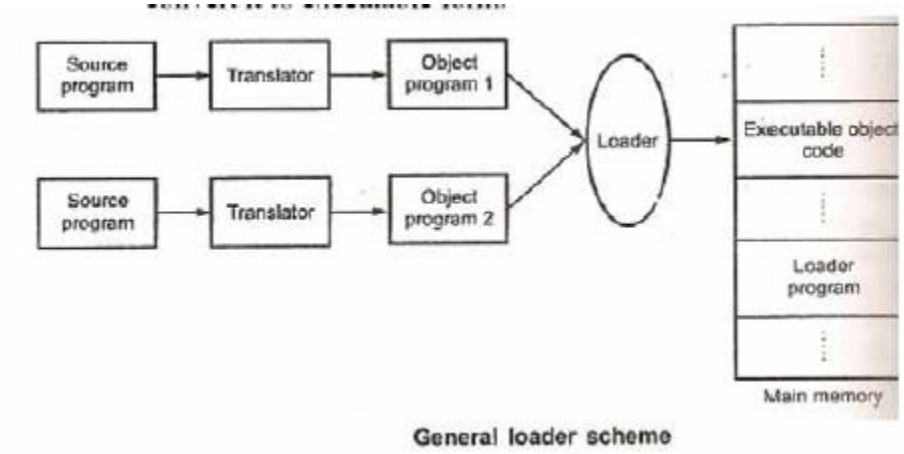
1. This scheme is easy to implement,
2. the assembler simply places the code into core and the loader, which consists of one instruction, transfers control to the starting instruction of the newly assembled program.

Disadvantages

1. In this scheme, a portion of memory is wasted. This is mainly because the core occupied by the assembler is not available to the object program.
2. It is essential to assemble the user's program deck every time it is executed.
3. It is quite difficult to handle multiple segments, if the source programs are in different languages. This disadvantage makes it difficult to produce orderly modular programs.

2. General Loader Scheme.

The concept of loaders can be well understood if one knows the general loader scheme. It is recommended for the general loader scheme that the instructions and data should be produced in the output, as they were assembled. This strategy, if followed, prevents the problem of wasting core for an assembler. When the code is required to be executed, the output is saved and loaded in the memory. The assembled program is loaded into the same area in core that it occupied earlier. The output that contains a coded form of the instructions is called the object deck. The object deck is used as intermediate data to avoid the circumstances in which the addition of a new program to a system is required. The loader accepts the assembled machine instructions, data and other information present in the object. The loader places the machine instructions and data in core in an executable computer form. More memory can be made available to a user, since in this scheme, the loader is assumed to be smaller than the assembler. Figure shows the general loader scheme.



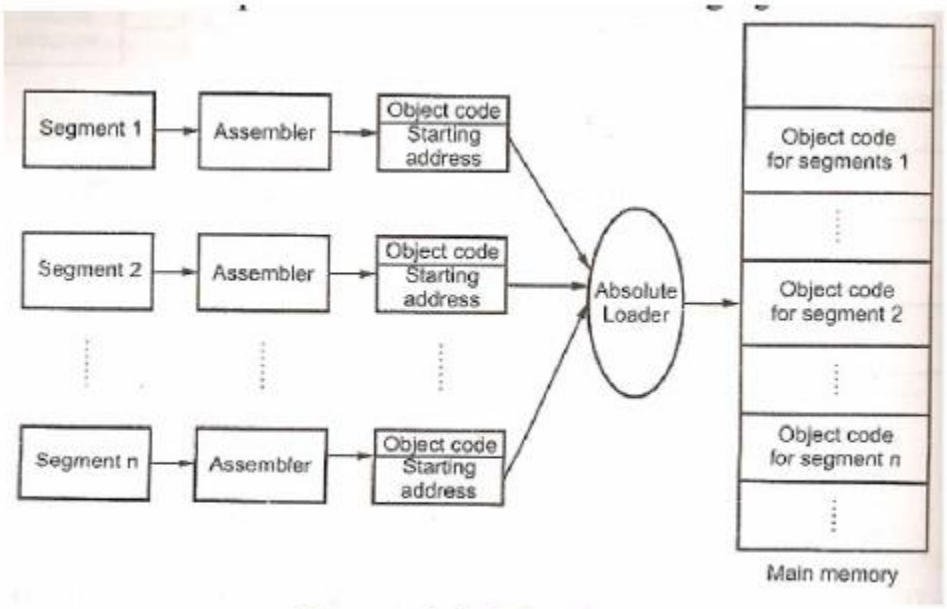
General loader scheme

Advantages:

- The program need not be retranslated each time while running it. This is because initially when source program gets executed an object program gets generated. Of program is not modified, and then loader can make use of this object program to convert it to executable form.
- There is no wastage of memory, because assembler is not placed in the memory, instead of it, loader occupies some portion of the memory. And size of loader is smaller than assembler, so more memory is available to the user.
- It is possible to write source program with multiple programs and multiple languages, because the source programs are first converted to object programs always, and loader accepts these object modules to convert it to executable form.

3. **Absolute Loader**

An absolute loader is the simplest type of loader scheme that fits the general model of loaders. The assembler produces the output in the same way as in the "compile and go loader". The assembler outputs the machine language translation of the source program. The difference lies in the form of data, i.e., the data in the absolute loader is punched on cards or you can say that it uses object deck as an intermediate data. The loader in turn simply accepts the machine language text and places it at the location prescribed by the assembler. When the text is being placed into the core, it can be noticed that much core is still available to the user. This is because, within this scheme, the assembler is not in the memory at the load time.



Process of absolute loading

In the figure, the MAIN program is assigned to locations 1000-2470 and the SQRT subroutine is assigned locations 4000-4770. This means the length of MAIN has increased to more than 3000 bytes, as it can be noticed from figure 4.4. If the modifications are required to be made in MAIN subroutine, then the end of MAIN subroutine, i.e., 1000+3000=4000, gets overlapped with the start of SQRT, i.e., with 4000. Therefore, it is necessary to assign a new location to SQRT. This can be made possible by changing the START pseudo-op card and

reassembling it. It is then quite obvious to modify all other subroutines that refer to address of SQRT.

Advantages

1. Absolute loaders are simple to implement.
2. This scheme allows multiple programs or the source programs written different languages. If there are multiple programs written in different languages then the respective language assembler will convert it to the language and a common object file can be prepared with all the ad resolution.
3. The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory.
4. The process of execution is efficient.

Disadvantages.

1. It is desirable for the programmer to specify the address in core where the program is to be loaded.
2. A programmer needs to remember the address of each subroutine, if there are multiple subroutines in the program.
3. Additionally, each absolute address is to be used by the programmer explicitly in the other subroutines such that subroutine linkage can be maintained.

4. Relocating Loaders

Relocating loaders was introduced in order to avoid possible reassembling of all subroutines when a single subroutine is changed. It also allows you to perform the tasks of allocation and linking for the programmer. The example of relocating loaders includes the Binary Symbolic Subroutine (BSS) loader. Although the BSS loader allows only one common data segment, it allows several procedure segments. The assembler in this type of loader assembles each procedure segment independently and passes the text and information to relocation and intersegment references.

In this scheme, the assembler produces an output in the form of text for each source program. A transfer vector that contains addresses, which includes names of the subroutines referenced by the source program, prefixes the output text. The assembler would also provide the loader with additional information such as the length of the entire program and also the length of the transfer vector portion. Once this information is provided, the text and the transfer vector get loaded into the core. Followed by this, the loader would load each subroutine, which is being identified in the transfer vector. A transfer instruction would then be placed to the corresponding subroutine for each entry in the transfer vector.

The output of the relocating assembler is the object program and information about all the programs to which it references. Additionally, it also provides relocation information for the locations that need to be changed if it is to be loaded in the core. This location may be arbitrary in

the core, let us say the locations, which are dependent on the core allocation. The BSS loader scheme is mostly used in computers with a fixed-length direct-address instruction format. Consider an example in which the 360 RX instruction format is as follows:

In this format, A2 is the 16-bit absolute address of the operand, which is the direct address instruction format. It is desirable to relocate the address portion of every instruction. As a result, the computers with a direct-address instruction format have much severe problems than the computes having 360-type base registers. The 360- type base registers solve the problem using relocation bits. The relocation bits are included in the object desk and the assembler associates a bit with each instruction or address field. The corresponding address field to each instruction must be relocated if the associated bit is equal to one; otherwise this field is not relocated.

5. Direct-Linking Loaders

A direct-linking loader is a general relocating loader and is the most popular loading scheme presently used. This scheme has an advantage that it allows the programmer to use multiple procedure and multiple data segments. In addition, the programmer is free to reference data or instructions that are contained in other segments. The direct linking loaders provide flexible intersegment referencing and accessing ability. An assembler provides the following information to the loader along with each procedure or data segment.

This information includes:

- Length of segment.
- List of all the symbols and their relative location in the segment that are referred by other segments.
- Information regarding the address constant which includes location in segment and description about the revising their values.
- Machine code translation of the source program and the relative addresses assigned.

LINKAGE EDITOR

Supply information needed to allow references between them. A linkage editor is also known as linker. To allow linking in a program, you need to perform:

- Program relocation
- Program linking

1. Program relocation

Program relocation is the process of modifying the addresses containing instructions of a program. You need to use program relocation to allocate a new memory address to the instruction. Instructions are fetched from the memory address and are followed sequentially to execute a program. The relocation of a program is performed by a linker and for performing relocation you need to calculate the relocation_factor that helps specify the translation time address in every instruction. Let the translated and linked origins of a program P be t_origin and l

The algorithm that you use for program relocation is:

1. program_linked_origin:=<link origin> from linker command;
2. For each object module
3. t_origin :=translated origin of the object module;
   OM_size :=size of the object module;
4. relocation_factor :=program_linked_origin-t_origin;
5. Read the machine language program in work_area;
6. Read RELOCTAB of the object module
7. For each entry in RELOCTAB
   A. **translated_addr**: = address in the RELOCTAB entry;
   B. **address_in_work**: =address of work_area + translated_address – t_origin;
   C. add relocation_factor to the operand in the wrd with the address address_in_work_area.
   D. **program_linked_origin**: = program_linked_origin + OM_size;

2. Program Linking

Linking in a program is a process of binding an external reference to a correct link address. You need to perform linking to resolve external reference that helps in the execution of a program. All the external references in a program are maintained in a table called name table (NTAB), which contains the symbolic name for external references or an object module. The information specified in NTAB is derived from
LINKTAB entries having type=PD. The algorithm that you use for program linking is:

Algorithm (Program Linking)

1. **program_linked_origin**: =<link origin> from linker command.
2. for each object module
   A. **t_origin**: =translated origin of the object module;
      OM_size :=size of the object module;
   B. **relocation_factor**: =program_linked_origin – t_origin;
   C. read the machine language program in work_area.
   D. Read LINKTAB of the object module.
   E. For each LINKTAB entry with type=PD
      name: =symbol;
      linked_address: =translated_address + relocation_factor;
      Enter (name, linked_address) in NTAB.
   F. Enter (object module name, program_linked_origin) in NTAB.
   G. **Program_linked_origin**: = program_linked_origin + OM_size;
3. for each object module
   A. **t_origin**: =translated origin of the object module;
      program_linked_origin :=load_address from NTAB;
   B. for each LINKTAB entry with type=EXT
      - **address_in_work_area**: =address of work_area + program_linked_origin - <link origin> + translated address – t_origin
      - search symbol in NTAB and copy its linked address. Add the linked address to the operand address in the word with the address address_in_work_area.

DYNAMIC LINKING

Sophisticated operating systems, such as Windows allow you to link executable object modules to be linked to a program while a program is running. This is known as dynamic linking. The operating system contains a linker that determines functions, which are not specified in a program. A linker searches through the specified libraries for the missing function and helps extract the object modules containing the missing functions from the libraries. The libraries are constructed in a way to be able to work with dynamic linkers. Such libraries are known as dynamic link libraries (DLLs). Technically, dynamic linking is not like static linking, which is done at build time. DLLs contain functions or routines, which are loaded and executed when needed by a program. The advantages of DLLs are:

- **Code sharing**: Programs in dynamic linking can share an identical code instead of creating an individual copy of a same library. Sharing allows executable functions and routines to be shared by many application programs. For example, the object linking and embedding (OLE) functions of OLE2.DLL can be invoked to allow the execution of functions or routines in any program.

- **Automatic updating**: Whenever you install a new version of dynamic link library, the older version is automatically overridden. When you run a program the updated version of the dynamic link library is automatically picked.

- **Securing**: Splitting the program you create into several linkage units makes it harder for crackers to read an executable file.

Unit 4

Unit -4

Common Object File Format (COFF)

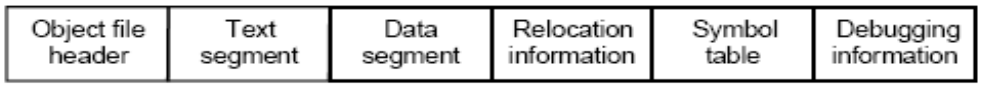
This chapter describes the Common Object File Format (COFF). COFF is the format of the output file produced by the assembler and the link editor.

The following are some key features of COFF:

- applications can add system-dependent information to the object file without causing access utilities to become obsolete
- space is provided for symbolic information used by debuggers and other applications
- programmers can modify the way the object file is constructed by providing directives at compile time

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains:

Object file format



- Object file header describes the size and position of the other pieces of the file
- Text segment contains the machine instructions
- Data segment contains binary representation of data in assembly file
- Relocation info identifies instructions and data that depend on absolute addresses
- Symbol table associates addresses with external labels and lists unresolved references
- Debugging info

Object file format

FILE HEADER
Optional Information
Section 1 Header
...
Section ***n*** Header
Raw Data for Section 1
...
Raw Data for Section ***n***
Relocation Info for Sect. 1
...
Relocation Info for Sect. ***n***
Line Numbers for Sect. 1
...
Line Numbers for Sect. ***n***
SYMBOL TABLE
STRING TABLE

The last four sections (relocation, line numbers, symbol table, and the string table) may be missing if the program is linked with the **-s** option of the **ld** command, or if the line number information, symbol table, and string table are removed by the command. The line number information does not appear unless the program is compiled with . Also, if there are no unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

File header

The file header contains the 20 bytes of information shown in. The last 2 bytes are flags that are used by **ld** and object file utilities.

File header contents

Bytes	Declaration	Name	Description
0-1	unsigned short	f_magic	Magic number
2-3	unsigned short	f_nscns	Number of sections
4-7	long int	f_timdat	Time and date stamp indicating when the file was created, expressed as the number of elapsed seconds since 00:00:00 GMT, January 1, 1970
8-11	long int	f_symptr	File pointer containing the starting address of the symbol table

12-15	long int	f_nsyms	Number of entries in the symbol table
16-17	unsigned short	f_opthdr	Number of bytes in the optional header
18-19	unsigned short	f_flags	Flags (see ``File header flags'')

Magic numbers

The magic number specifies the target machine on which the object file is executable.

Flags

The last 2 bytes of the file header are flags that describe the type of the object file. Currently defined flags are found in the header file *filehdr.h* and are shown in ``File header flags''.

File header flags

Mnemonic	**Flag**	**Meaning**
F_RELFLG	00001	Relocation information stripped from the file
F_EXEC	00002	File is executable (i.e., no unresolved external references)
F_LNNO	00004	Line numbers stripped from the file
F_LSYMS	00010	Local symbols stripped from the file
F_AR16WR	0000200	16-bit byte reversed word
F_AR32WR	0000400	32-bit byte reversed word

File header declaration

The C structure declaration for the file header is shown below. This declaration may be found in the header file *filehdr.h*.

Section headers

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in.

Section header contents

Bytes	Declaration	Name	Description
0-7	char	s_name	8-character null padded section name
8-11	long int	s_paddr	Physical address of section
12-15	long int	s_vaddr	Virtual address of section
16-19	long int	s_size	Section size in bytes
20-23	long int	s_scnptr	File pointer to raw data
24-27	long int	s_relptr	File pointer to relocation entries
28-31	long int	s_lnnoptr	File pointer to line number entries
32-33	unsigned short	s_nreloc	Number of relocation entries
34-35	unsigned short	s_nlnno	Number of line number entries
36-39	long int	s_flags	Flags (see ``Section header flags'')

The size of a section is padded to a multiple of 4 bytes. File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the function **fseek**(S).

Line numbers

When invoked with **cc -g** the compiler causes an entry in the object file for every source line where a breakpoint can be inserted. You can then reference line numbers when using a software debugger like **sdb**(CP). All line numbers in a section are grouped by function as shown in ``Line number grouping''.

Line number grouping

symbol index	0
physical address	line number
physical address	line number
.	.
.	.
.	.
symbol index	0
physical address	line number
physical address	line number

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries are relative to the beginning of the function and appear in increasing order of address.

Line number declaration

The structure declaration currently used for line number entries is shown below.

```
struct lineno
{
   union
   {
         long  l_symndx;  /* symtbl index of func name */
         long  l_paddr;   /* paddr of line number */
   } l_addr;
   unsigned short  l_lnno;    /* line number */
};
```

Symbol table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown in ``COFF symbol table''.

COFF symbol table

filename 1
function 1
C_WEAKEXT aliases
for function 1
function 1b (alias)
. . .
local symbols for function 1
function 2
C_WEAKEXT aliases
for function 2

. . .
local symbols for function 2
. . .
statics
. . .
filename 2
function 1
C_WEAKEXT aliases
for function 1
. . .
local symbols for function 1
. . .
statics
. . .
defined global symbols
undefined global symbols

The word ``statics'' in ``COFF symbol table'' means symbols defined with the C language storage class static outside any function. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

String table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table, therefore, are greater than or equal to 4. For example, given a file containing two symbols (with names longer then eight characters, ***long_name_1*** and ***another_one***) the string table has the format as shown in ``String table'':

String table

`l'	`o'	`n'	`g'
`_'	`n'	`a'	`m'
`e'	`_'	`l'	`\0'
`a'	`n'	`o'	`t'
`h'	`e'	`r'	`_'
`o'	`n'	`e'	`\0'

The index of ***long_name_1*** in the string table is 4 and the index of ***another_one*** is 16.

Debugger

A **debugger** or **debugging tool** is a computer program that is used to test and debug other programs (the "target" program). The code to be examined might alternatively be running on an *instruction set simulator* (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered but which will typically be somewhat slower than executing the code directly on the appropriate (or the same) processor. Some debuggers offer two modes of operation—full or partial simulation—to limit this impact.

A "trap" occurs when the program cannot normally continue because of a programming bug or invalid data. For example, the program might have tried to use an instruction not available on the current version of the CPU or attempted to access unavailable or protected memory. When the program "traps" or reaches a preset condition, the debugger typically shows the location in the original code if it is a **source-level debugger** or **symbolic debugger**, commonly now seen in integrated development environments. If it is a **low-level debugger** or a **machine-language debugger** it shows the line in the disassembly

A debugger is a program that runs other programs, allowing the user to exercise control over these programs, and to examine variables when problems arise.

GNU Debugger which is called **gdb** is the most popular debugger for UNIX systems to debug C and C++ Programs.

GNU Debugger helps you in finding out followings:

- If a core dump happened then what statement or expression did the program crash on?
- If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters?
- What are the values of program variables at a particular point during execution of the program?
- What is the result of a particular expression in a program?

How GDB Debugs?

GDB allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

GDB uses a simple command line interface.

Note the Followings:

- Even though GDB can help you in finding out memory leakage related bugs but it is not a tool to detect memory leakages
- GDB cannot be used for programs that do not compile without errors and it does not help in fixing those errors.

Unit 5

Unit -5

Device driver (commonly referred to as a *driver*) is a computer program that operates or controls a particular type of device that is attached to a computer. A driver provides a software interface to hardware devices, enabling operating systems and other computer

UNIT-5

UNIT-5

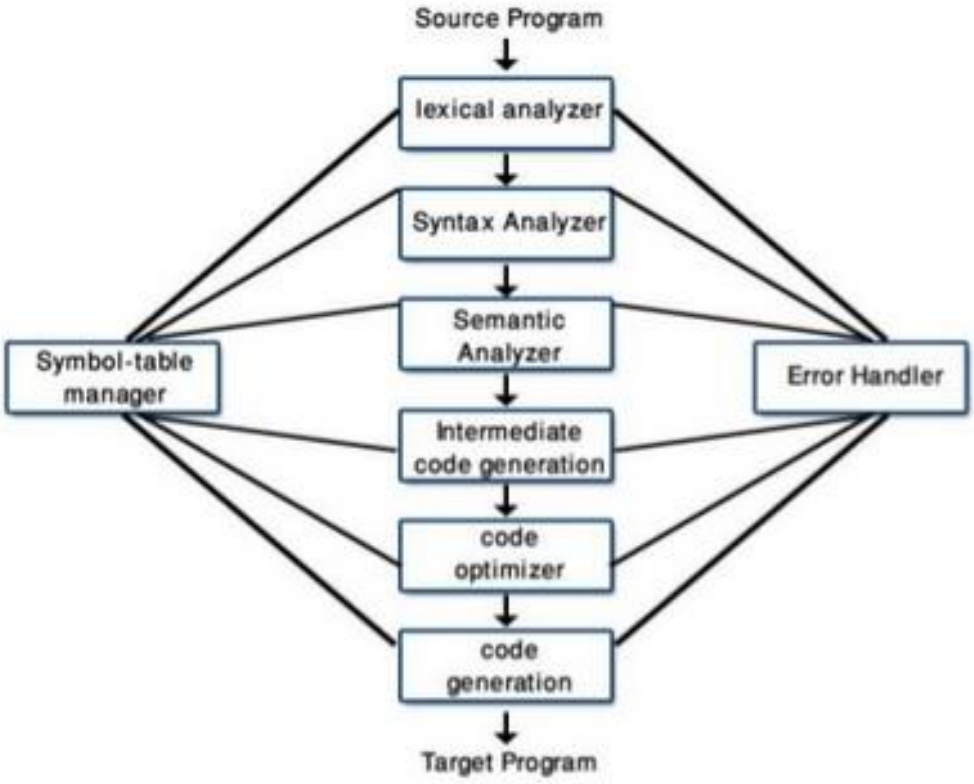
Compiler

The word *compilation* is used to denote the task of translating *high level language* (HLL) programs into machine language programs. Though the objective of this task of translation is similar to that of an *assembler*, the problem of compilation is much more complex than that of an assembler. A *compiler* is a program that does the compilation task. A compiler recognizes programs in a particular HLL and produces equivalent output programs appropriate for some particular computer configuration (hardware and OS). Thus, an HLL program is to a great extent independent of the configuration of the machine it will eventually run on, as long as it is ensured that the program is compiled by a compiler that recognizes that HLL and produces output for the required machine configuration. It is common for a machine to have compilers that would translate programs to produce executables for that machine (*hosts*). But there also are compilers that runs on one type of machine but the output of which are programs that shall run on some other machine configuration, such as generating an MS-DOS executable program by compiling an HLL program in UNIX. Such a compiler is called a *cross compiler*. Another kind of translator that accepts programs in HLL are known as *interpreters*. An interpreter translates an input HLL program and *also runs the program on the same machine*. Hence the output of running an interpreter is actually the output of the program that it translates.

Important phases in Compilation

The following is a typical breakdown of the overall task of a compiler in an approximate sequence -

- Lexical analysis
- Syntax analysis
- Intermediate code generation
- Code optimization
- Code generation.



Like an assembler, a compiler usually performs the above tasks by making multiple passes over the input or some intermediate representation of the same. The compilation task calls for intensive processing of information extracted from the input programs, and hence data structures for representing such information need to be carefully selected. During the process of translation a compiler also detects certain kinds of errors in the input, and may try to take some recovery steps for these.

Lexical Analysis

Lexical analysis in a compiler can be performed in the same way as in an assembler. Generally in an HLL there are more number of tokens to be recognized - various keywords (such as, *for, while, if, else,* etc.), punctuation symbols (such as, comma, semi-colon, braces, etc.), operators (such as arithmetic operators, logical operators, etc.), identifiers, etc. Tools like *lex* or *flex* are used to create lexical analysers.

Syntax Analysis

Syntax analysis deals with recognizing the structure of input programs according to known set of *syntax rules* defined for the HLL. This is the most important aspect in which HLLs are significantly different from lower level languages such as assembly language. In assembly languages the syntax rules are simple which roughly requires that a program should be a sequence of statements, and each statement should essentially contain a mnemonic followed by zero or more operands depending on the mnemonic. Optionally, there can be also being an identifier preceding the mnemonic. In case of HLLs, the syntax rules are much more complicated. In most HLLs the notion of a

statement itself is very flexible, and often allows *recursion*, making nested constructs valid. These languages usually support multiple data types and often allow programmers to define abstract data types to be used in the programs. These and many other such features make the process of creating software easier and less error prone compared to assembly language programming. But, on the other hand, these features make the process of compilation complicated.

The non-trivial syntax rules of HLLs need to be cleverly specified using some suitable notation, so that these can be encoded in the compiler program. One commonly used formalism for this purpose is the *Context Free Grammar (CFG)*. CFG is a formalism that is more powerful than *regular grammars* (used to write regular expressions to describe *tokens* in a lexical analyser). Recursion, which is a common feature in most constructs of HLLs, can be defined using a CFG in a concise way, whereas a regular grammar is incapable of doing so. It needs to be noted that there are certain constructs that cannot be adequately described using CFG, and may require other more powerful formalisms, such as *Context Sensitive Grammars (CSG)*. A common notation used to write the rules of CFG or CSG is the *BNF (Backus Naur Form).*

During syntax analysis, the compiler tries to apply the rules of the grammar of the input HLL given using BNF, to recognise the structure of the input program. This is called *parsing* and the module that performs this task is called a *parser*. From a somewhat abstract point of view, the output of this phase is a *parse tree* that depicts how various rules of the grammar can be repetitively applied to recognise the input program. If the parser cannot create a parse tree for some given input program, then the input program is not valid according to the syntax of the HLL.

The soundness of the CFG formalism and the BNF notation makes it possible to create different types of efficient parsers to recognise input according to a given language. These parsers can be broadly classified as *top-down parsers* and *bottom-up parsers*. *Recursive descent parsers* and *Predictive parsers* are two examples of top-down parsers. *SLR parsers* and *LALR parser* are two examples of bottom-up parsers. For certain simple context free languages (languages that can be defined using CFG) simpler bottom-up parsers can be written. For example, for recognising mathematical expressions, an *operator precedence parser* can be created.

In creating a compiler, a parser is often built using tools such as *yacc* and *bison*. To do so the CFG of the input language is written in BNF notation, and given as input to the tool (along with other details).

Intermediate Code Generation

Having recognized a given input program as valid, a compiler tries to create the equivalent program in the language of the target environment. In case of an assembler this translation was somewhat simpler since the operation implied by the mnemonic opcode in each statement in the input program, there is some equivalent machine opcode. The number of operands applicable for each operation in the machine language is the same as allowed for the corresponding assembly language mnemonic opcodes. Thus for the assembly language the translation for each statement can be done for each statement almost independently of the rest of the program. But, in case of an HLL, it is futile to try to associate a single machine opcode for each statement of the input language. One of the

reasons for this is, as stated above, the extent of a statement is not always fixed and may contain recursion. Moreover, data references in HLL programs can assume significant levels of abstractions in comparision to what the target execution environment may directly support. The task of associating meanings (in terms of primitive operations that can be supported by a machine) to programs or segments of a program is called *semantic processing*.

Though it is not entirely straightforward to associate target language operations to statements in the HLL programs, the CFG for the HLL allows one to associate *semantic actions* (or implications) for the various syntactic rules. Hence in the broad task of translation, when the input program is parsed, a compiler also tries to perform certain semantic actions corresponding to the various syntactic rules that are eventually applied. However, most HLLs contain certain syntactic features for which the semantic actions are to be determined using some additional information, such as the contents of the symbol table. Hence, building and usage of data-structures such as the symbol table are an important part of the semantic action that are performed by the compiler.

Upon carrying out the semantic processing a more manageable equivalent form of the input program is obtained. This is stored (represented) using some *Intermediate code* representation that makes further processing easy. In this representation, the compiler often has to introduce several temporary variables to store intermediate results of various operations. The language used for the intermediate code is generally not any particular machine language, but is such which can be efficiently converted to a required machine language (some form of assembly language can be considered for such use).

Code Optimisation

The programs represented in the intermediate code form usually contains much scope for optimization both in terms of storage space as well as run time efficiency of the intended output program. Sometimes the input program itself contains such scope. Besides that, the process of generating the intermediate code representation usually leaves much room for such optimization. Hence, compilers usually implement explicit steps to optimise the intermediate code.

Code Generation

Finally, the compiler converts the (optimised) program in the intermediate code representation to the required machine language. It needs to be noted that if the program being translated by the compiler actually has dependencies on some external modules, then *linking* has to be performed to the output of the compiler. These activities are independent of whether the input program was in HLL or assembly language.

Lexical analysis

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called

a **lexical analyzer**, **lexer**, or **scanner**. A lexer often exists as a single function which is called by a **parser** or another function.

Role of the lexical analyzer

The main task is to read the input characters and produce as output sequence of tokens that the parser uses for syntax analysis.

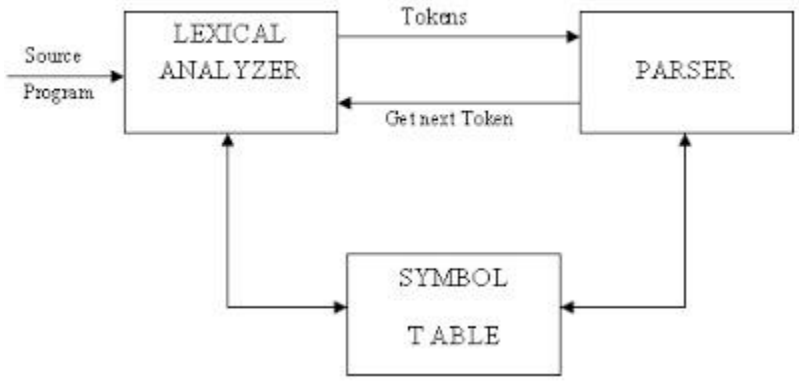


Fig 2.1 role of the lexical analyzer diagram

Up on receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.
Its secondary tasks are,

- One task is stripping out from the source program comments and white space is in the form of blank, tab, new line characters.
- Another task is correlating error messages from the compiler with the source program.

Sometimes lexical analyzer is divided in to cascade of two phases.

1) Scanning
2) lexical analysis.

The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

Stmt →if expr
then stmt
| If expr then else stmt
| ϵ
Expr →term relop term
| term
Term →id
|number

For relop ,we use the comparison operations of languages like Pascal or SQL where = is "equals" and < > is "not equals" because it presents an interesting structure of lexemes. The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

digit -->[0,9]
digits --
>digit+
number -->digit(.digit)?(e.[+-]?digits)?
letter -->[A-Z,a-z]
id --
>letter(letter/digit)*
if -->
if
then -->then
else -->else
relop --></>/<=/>=/==/< >

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

ws →(blank/tab/newline)$^+$

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

Lexeme	**Token Name**	**Attribute Value**
Any ws	_	_
if	if	_
then	then	_
else	else	_
Any id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	ET
< >	relop	NE

Transition Diagram:

Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols.

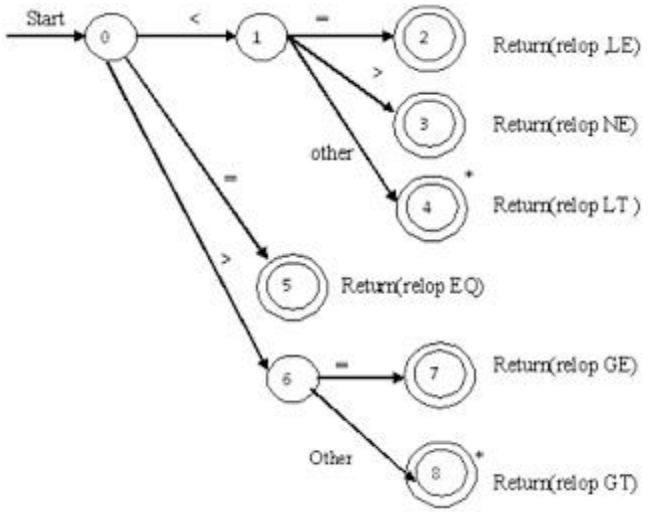
If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.

2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.

3. One state is designed the state ,or initial state ., it is indicated by an edge labeled "start" entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.



Theory

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so 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. Initially we will simply print the matched string rather than return a token value.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

letter(letter|digit)*

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

repetition, expressed by the "*" operator alternation, expressed by the "|" operator concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.

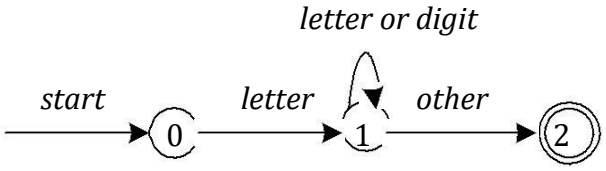


Figure 3: Finite State Automaton

In Figure 3 state 0 is the start state and state 2 is the accepting state. As characters are read we make a transition from one state to another. When the first letter is read we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit we transition to accepting state 2. *Any* FSA may be expressed as a computer program. For example, our 3-state machine is easily programmed:

```
  start:  goto state0
state0: read c
            if c = letter goto state1
          goto state0
state1: read c
            if c = letter goto state1
            if c = digit goto state1
         goto state2


state2: accept string
```

This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next *input* character and *current state* the next state is easily determined by indexing into a computer-generated state table.

Now we can easily understand some of lex's limitations. For example, lex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a " **(**" we push it on the stack. When a "**)**" is encountered we match it with the top of the stack and pop the stack. However lex only has states and transitions between states. Since it has no stack it is not well suited for parsing nested structures. Yacc augments an FSA with a stack and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching. Yacc is appropriate for more challenging tasks.

Practice

Metacharacter	**Matches**
.	any character except newline
\n	newline
*****	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	**a** or **b**
(ab)+	one or more copies of **ab** (grouping)
"a+b"	literal " " (C escapes still work)
	a+b
[]	character class

Table 1: Pattern Matching Primitives

Expression	**Matches**
abc	**abc**
abc*	**ab abc abcc abccc ...**
abc+	**abc abcc abccc ...**
a(bc)+	**abc abcbc abcbcbc ...**
a(bc)?	**a abc**
[abc]	one of: **a b c**
[a-z]	any letter, a-z
[a\-z]	one of: **a - z**
[-az]	one of: **a - z**

	of:
	- a z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: **a b**
[a^b]	one of: **a ^ b**
[a|b]	one of: **a | b**
a|b	**a b**

Table 2: Pattern Matching Examples

Regular expressions in lex are composed of metacharacters (Table 1). Pattern-matching examples are shown in Table 2. Within a character class normal operators lose their meaning.

Two operators allowed in a character class are the hyphen ("**-**") and circumflex ("**^**"). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Input to Lex is divided into three sections with **%%** dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

```
%%
```

Input is copied to output one character at a time. The first **%%** is always required, as there must always be a rules section. However if we don't specify any rules then the default action is to match everything and copy it to output. Defaults for input and output are **stdin** and **stdout**, respectively. Here is the same example with defaults explicitly coded:

```
%%
        /* match everything except newline */
.       ECHO;
        /* match newline */
\n ECHO;
%%

int yywrap(void) {
        return 1;
}

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

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline) and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements, enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, "**.**" and "**\n**", with an **ECHO** action associated for each pattern. Several macros and variables are predefined by lex. **ECHO** is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, **ECHO** is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable **yytext** is a pointer to the matched string (NULL-terminated) and **yyleng** is the length of the matched string. Variable **yyout** is the output file and defaults to stdout. Function **yywrap** is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a **main** function. In this case we simply call **yylex** that is the main entry-point for lex. Some implementations of lex include copies of **main** and **yywrap** in a library thus eliminating the need to code them explicitly. This is why our first example, the shortest lex program, functioned properly.

Name	Function
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN	condition switch start condition
ECHO	write matched string

Table 3: Lex Predefined Variables

Here is a program that does nothing at all. All input is matched but no action is associated with any pattern so there will be no output.

```
%%
.
\n
```

The following example prepends line numbers to each line in a file. Some implementations of lex predefine and calculate **yylineno**. The input file for lex is **yyin** and defaults to **stdin**.

```
%{
    int yylineno;
%}
%%
^(.*)\n      printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) { yyin
    = fopen(argv[1], "r"); yylex();
    fclose(yyin);
}
```

The definitions section is composed of substitutions, code, and start states. Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with "**%{**" and "**%}**" markers. Substitutions simplify pattern-matching rules. For example, we may define digits and letters:

```
digit [0-9] letter
[A-Za-z] %{
    int count;
%}
%%
    /* match identifier */
{letter}({letter}|{digit})* count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

Whitespace must separate the defining term and the associated expression. References to substitutions in the rules section are surrounded by braces (**{letter}**) to distinguish them from literals. When we have a match in the rules section the associated C code is executed. Here is a scanner that counts the number of characters, words, and lines in a file (similar to Unix wc):

```
%{
    int nchar, nword, nline;
%}
%%
\n          { nline++; nchar++; }
[^ \t\n]+   { nword++, nchar += yyleng; }
.           { nchar++; }
%%
int main(void) {
    yylex();
    printf("%d\t%d\t%d\n", nchar, nword, nline);
    return 0;
}
```

Yacc

Theory

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique, pioneered by John Backus and Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express *context-free* languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

E -> E + E
E -> E * E
E -> id

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as **E** (expression) are nonterminals. Terms such as **id** (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

E -> E * E	**(r2)**
-> E * z	**(r3)**
-> E + E * z	**(r1)**
-> E + y * z	**(r3)**
-> x + y * z	**(r3)**

At each step we expanded a term and replace the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression we need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar we need to reduce an expression to a single nonterminal. This is known as *bottom-up* or *shift-reduce* parsing and uses a stack for storing terms. Here is the same derivation but in reverse order:

1	**. x + y * z**	**shift**	
2	**x . + y * z**	**reduce(r3)**	
3	**E . + y * z**	**shift**	
4	**E + . y * z**	**shift**	
5	**E + y . * z**	**reduce(r3)**	
6	**E + E . * z**	**shift**	
7	**E + E * . z**	**shift**	
8	**E + E * z .**	**reduce(r3)**	
9	**E + E * E .**	**reduce(r2)**	**emit multiply**
10	**E + E .**	**reduce(r1)**	**emit add**
11	**E .**	**accept**	

Terms to the left of the dot are on the stack while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production we replace the matched tokens on the stack with the lhs of the production. In other words the matched tokens of the rhs are popped

off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a *handle* and we are *reducing* the handle to the lhs of the production. This process continues until we have shifted all input to the stack and only the starting nonterminal remains on the stack. In step 1 we shift the **x** to the stack. Step 2 applies rule r3 to the stack to change **x** to **E**. We continue shifting and reducing until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply

1
1

instruction. Similarly the add instruction is emitted in step 10. Consequently multiply has a higher precedence than addition.

Consider the shift at step 6. Instead of shifting we could have reduced and apply rule r1. This would result in addition having a higher precedence than multiplication. This is known as a *shift-reduce* conflict. Our grammar is *ambiguous* because there is more than one possible derivation that will yield the expression. In this case operator precedence is affected. As another example, associativity in the rule

E -> E + E

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar or supply yacc with directives that indicate which operator has precedence. The latter method is simpler and will be demonstrated in the practice section.

The following grammar has a *reduce-reduce* conflict. With an **id** on the stack we may reduce to **T**, or **E**.

E -> T
E -> id
T -> id

Yacc takes a default action when there is a conflict. For shift-reduce conflicts yacc will shift. For reduce-reduce conflicts it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.

Practice, Part I

... definitions ...
%%
... rules ...
%%
... subroutines ...

Input to yacc is divided into three sections. The definitions section consists of token declarations and C code bracketed by " **%{**" and "**%}**". The BNF grammar is placed in the rules section and user subroutines are added in the subroutines section.

This is best illustrated by constructing a small calculator that can add and subtract numbers. We'll begin by examining the linkage between lex and yacc. Here is the definitions section for the yacc input file:

%token INTEGER

This definition declares an **INTEGER** token. Yacc generates a parser in file **y.tab.c** and an include file, **y.tab.h**:

#ifndef YYSTYPE

```
#define YYSTYPE int
#endif
#define INTEGER 258

extern YYSTYPE yylval;
```

1
2

Lex includes this file and utilizes the definitions for token values. To obtain tokens yacc calls **yylex** . Function **yylex** has a return type of int that returns a token. Values associated with the token are returned by lex in variable **yylval**. For example,

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

would store the value of the integer in **yylval** , and return token **INTEGER** to yacc. The type of **yylval** is determined by **YYSTYPE**. Since the default type is integer this works well in this case. Token values 0-255 are reserved for character values. For example, if you had a rule such as

```
[-+]            return *yytext;          /* return operator */
```

the character value for minus or plus is returned. Note that we placed the minus sign first so that it wouldn't be mistaken for a range designator. Generated token values typically start around 258 because lex reserves several values for end-of-file and error processing. Here is the complete lex input specification for our calculator:

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h" %}
%%

[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }

[-+\n]      return *yytext;

[ \t]       ; /* skip whitespace */

.           yyerror("invalid character");

%%

int yywrap(void) {
    return 1;
}
```

Internally yacc maintains two stacks in memory; a parse stack and a value stack. The parse stack contains terminals and nonterminals that represent the current parsing state. The value stack is an array of **YYSTYPE** elements and associates a value with each element in the parse stack. For example when lex returns an **INTEGER** token yacc shifts this token to the parse stack. At the same time the corresponding **yylval** is shifted to the value stack. The parse and value stacks are always synchronized so

finding a value related to a token on the stack is easily accomplished. Here is the yacc input specification for our calculator:

1
3

```
%{
    #include <stdio.h> int
    yylex(void); void
    yyerror(char *);
%}
%token INTEGER

%%
program:
        program expr '\n'           { printf("%d\n", $2); }
        |
        ;
expr:
        INTEGER                     { $$ = $1; }
        | expr '+' expr             { $$ = $1 + $3; }
        | expr '-' expr             { $$ = $1 - $3; }
        ;
%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}
int main(void) {
    yyparse();
    return 0;
}
```

The rules section resembles the BNF grammar discussed earlier. The left-hand side of a production, or nonterminal, is entered left-justified and followed by a colon. This is followed by the right-hand side of the production. Actions associated with a rule are entered in braces.

With left-recursion, we have specified that a program consists of zero or more expressions. Each expression terminates with a newline. When a newline is detected we print the value of the expression. When we apply the rule

```
expr: expr '+' expr                 { $$ = $1 + $3; }
```

we replace the right-hand side of the production in the parse stack with the left-hand side of the same production. In this case we pop "**expr '+' expr**" and push "**expr**". We have reduced the stack by popping three terms off the stack and pushing back one term. We may reference positions in the value stack in our C code by specifying "**$1**" for the first term on the right-hand side of the production, "**$2**" for the second, and so on. " **$$**" designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum. As a consequence the parse and value stacks remain synchronized.

1
4

Numeric values are initially entered on the stack when we reduce from **INTEGER** to **expr**. After **INTEGER** is shifted to the stack we apply the rule

expr: INTEGER { $$ = $1; }

The **INTEGER** token is popped off the parse stack followed by a push of **expr**. For the value stack we pop the integer value off the stack and then push it back on again. In other words we do nothing. In fact this is the default action and need not be specified. Finally, when a newline is encountered, the value associated with **expr** is printed.

In the event of syntax errors yacc calls the user-supplied function **yyerror**. If you need to modify the interface to **yyerror** then alter the canned file that yacc includes to fit your needs. The last function in our yacc specification is **main** … in case you were wondering where it was. This example still has an ambiguous grammar. Although yacc will issue shift-reduce warnings it will still process the grammar using shift as the default operation.